REPORT DOCUMENTATION PAGE			Form Approved OMB NO. 0704-0188
Public Reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comment regarding this burden estimates or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188,) Washington, DC 20503.			
1. AGENCY USE ONLY (Leave Blank)	2. REPORT DATE 15-May-02		3. REPORT TYPE AND DATES COVERED Final Thesis, August 2001 - June 2002
4. TITLE AND SUBTITLE Change detection in XML documents of differing levels of structural verbosity in support of ubiquitous data access			5. FUNDING NUMBERS
6. AUTHOR(S) Lanham, Michael J.			
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) University of Florida, Department of Computer and Information Sciences & Engineering, Gainesville, FL 32602			8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES) US Military Academy Department of Electrical Engineering and Computer Science West Point, NY 10996			10. SPONSORING / MONITORING AGENCY REPORT NUMBER
11. SUPPLEMENTARY NOTES The views, opinions and/or findings contained in this report are those of the author(s) and should not be construed as an official Department of the Army position, policy or decision, unless so designated by other documentation.			
12 a. DISTRIBUTION / AVAILABILITY STATEMENT Approved for public release; distribution unlimited.			12 b. DISTRIBUTION CODE
13. ABSTRACT (Maximum 200 words) Extensible markup Language (XML) and difference algorithms help mobile computing devices achieve interoperability with each other. XML provides applications a non-proprietary means of communicating among each other. Difference algorithms allow transferring only changes to files, not entire changed files. This research focuses on customizing current XML difference algorithms. The goal is to detect differences between two XML files that do not necessarily share the same structure. Small devices are generally incapable of presenting and processing parts of multi-media documents. They also have less memory and storage space than desktop units have. The sending device reduces transmission time by removing those contents, and other large embedded objects, from the data stream. The user can edit the streamlined file on the mobile device. Transmitting on the changes to this edited file conserves bandwidth, connection time, and battery-power. Integrating the transmitted changes back into the originating complex document becomes a fundamental requirement and a significant challenge.			
14. SUBJECT TERMS XML, change detection, ubiquitous data access			15. NUMBER OF PAGES 95
			16. PRICE CODE
17. SECURITY CLASSIFICATION OR REPORT **UNCLASSIFIED**	18. SECURITY CLASSIFICATION ON THIS PAGE **UNCLASSIFIED**	19. SECURITY CLASSIFICATION OF ABSTRACT **UNCLASSIFIED**	20. LIMITATION OF ABSTRACT **UL**

NSN 7540-01-280-5500

Standard Form 298 (Rev.2-89)
Prescribed by ANSI Std. 239-18
298-102

Enclosure 1

CHANGE DETECTION IN XML DOCUMENTS OF DIFFERING LEVELS OF STRUCTURAL VERBOSITY IN SUPPORT OF UBIQUITOUS DATA ACCESS

By

MICHAEL J. LANHAM

A THESIS PRESENTED TO THE GRADUATE SCHOOL
OF THE UNIVERSITY OF FLORIDA IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF ENGINEERING

UNIVERSITY OF FLORIDA, 2002

Approved for Public Release: Distribution Unlimited

Copyright 2002

By

Michael J. Lanham

I dedicate this work and this thesis to my wife, my son, and my family. I also wish to dedicate this work to the men and women of the United States Army—protectors and defenders of this great nation.

ACKNOWLEDGMENTS

All thanks that I offer are poor substitutes for the gratitude and appreciation that I owe to my family, my advisors, my friends, and the United States Army.

With the help of my wife and son, my life remains balanced and fulfilling to a degree I never dreamed possible. Adding to their support is the love and affection of my family: mother, father, brother, sister and their families. My family motivates me everyday with a thrill for life. Long after I depart this earth, my children and my family will remember what kind of man I was: may I live up to what they deserve.

I wish to thank Dr. Joachim Hammer and Dr. Abdelsalam Helal. Their co-sponsoring of the Ubiquitous Data Access project has been instrumental in my efforts at securing this step in my professional life. Their numerous insights, corrections, and guidance are directly responsible for any success I had in contributing to their project. Finally, I wish to thank Dr. Joseph Wilson, Jim Hranicky, and Sullivan Beck for their unending patience.

Finally I wish to thank the United States Army. The Army has given, and continues to give, emotional and job satisfaction that I could only have wished for. I am proud to have served next to the soldiers and officers of the United States Army. I am pleased the Army continues to let me fulfill my oath "to protect and defend the Constitution of the United States of America, against all enemies, foreign and domestic, so help me God."

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Abstract of Thesis Presented to the Graduate School
of the University of Florida in Partial Fulfillment of the
Requirements for the Degree of Master of Engineering

CHANGE DETECTION IN XML DOCUMENTS OF DIFFERING STRUCTURAL VERBOSITY IN SUPPORT OF UBIQUITOUS DATA ACCESS

By

Michael J. Lanham

August 2002

Chair: Dr. Joachim Hammer
Department: Computer and Information Sciences and Engineering

Mobile computing used to mean the Osborne 1 personal computer and twenty-four pounds of electronics in a hard plastic case. The world now has six ounce palm top computers and personal digital assistants (PDA) with more processing power than ten Osborne computers. Along with the increasing capabilities of mobile devices come increasing consumer expectations. People are not content to maintain separate electronic address lists and calendars on every device they own. Nor are they content to lose edits, formatting, and time when exporting and importing documents to and from applications with different formats. These rising expectations and the expanding capabilities of mobile computing devices intersect in the world of ubiquitous data access. Consumers want to be able to access and modify the same information on multiple devices, see those changes on every device, and do so without significant effort.

Through the use of Extensible Markup Language (XML) and intelligent difference algorithms, these and other mobile devices are within reach of achieving interoperability.

XML provides applications a non-proprietary means of communicating with each other. Difference algorithms allow devices with limited battery life and inconsistent-quality network connections to transfer modified data in small pieces. Combining XML with change detection algorithms may reduce the need for transferring entire files to and from devices.

This research focuses on customizing current XML difference algorithms. The goal is to detect differences between two XML files that do not always share the same structure, though they are directly related through their content. This is the case when a user has transferred a complex, multi-media document from his/her desktop personal computer to a palm-top device. The small device is incapable of presenting some of the document's contents and has less memory and storage space than the desktop unit. The sending device can reduce transmission time by removing those contents, and other large embedded objects, from the data stream. Now that the streamlined version of the document is on the small device, the user can edit the file using the software on that device. Transmitting only the changes in this edited file continues our effort to conserve bandwidth, connection time, and battery-power. Integrating the transmitted changes from the streamlined document back into the originating complex document becomes a fundamental requirement and a significant challenge.

Users want their various portable and non-portable devices to be able to share data. They want modifications made on one device to propagate to other devices they use. This research is one piece of the ongoing effort to achieve ubiquitous data access.

CHAPTER 1
INTRODUCTION

Motivation

Mobile computing used to mean the Osborne 1 [JON94] and twenty-four pounds of electronics in a hard plastic case. Now the world has six ounce palm top computers, personal digital assistants (PDA), and phones with more processing power than ten Osborn computers. Along with the increasing capabilities of mobile devices come the increasing expectations of consumers. People are no longer content to maintain three electronic address lists, two different electronic calendars, and have to work with three incompatible word processors. These rising expectations and the expanding capabilities of mobile computing devices intersect in the world of ubiquitous data access. Consumers want to be able to access and modify the same information on multiple devices, without a lot of effort. Interoperability of the various software packages on these mobile devices remains an elusive and unlikely prospect.

Through the use of Extensible Markup Language (XML) [WWW00] and intelligent use of difference algorithms, these and other mobile devices are within reach of achieving true interoperability. XML provides an open-standard means of communicating between applications that would otherwise not share data well. Difference algorithms allow devices with limited battery life and inconsistent-quality connections to transfer modified data in small pieces: eliminating the need for multi-minute transmission of data to and from a device.

This presentation focuses on customizing current XML difference algorithms. The goal is to detect differences between two XML files that do not always share the same structure. This is the case when a user has transferred a complex, multi-media document from his[1] desktop personal computer to his palm top device. The small device is incapable of presenting some of the document's content and has less memory and storage space than the desktop unit. The sending device can reduce transmission time by removing those contents, and other large embedded objects, from the data stream. Now that the streamlined version of the document is on the small device, the user can edit the file using software on the mobile device. Integrating the changes from this streamlined document back into the originating complex document is thus a requirement and a significant challenge.

Users want their various portable and non-portable devices to be able to share data. They want this sharing to be automatic and with little active intervention. Finally they want modifications made on one device to propagate to the other devices they use. This research is one piece of the ongoing effort to achieve ubiquitous data access.

Ubiquitous Data Access Project

The Ubiquitous Data Access Project (UbiData) at the University of Florida is a project operating under the guidance of Dr. Abdelsalam "Sumi" Helal and Dr. Joachim Hammer. Their work in [ZHA01, HEL01] states the project's goals simply while setting a high bar for expected performance levels. The goals of UbiData are as follows:

- Any time and anywhere access to data. Regardless of the connectivity status of a user's device, the user should be able to read and modify the data in his device. The immediate requirements are three modes of connectivity. The first is via a high-speed Internet connection. The second mode is weakly

[1] Throughout this paper, the terms he, his, etc. refer to an individual who may be male or female.

connected via a low-bandwidth high latency connection such as wireless LANs, cell phones, or telephone lines. The third mode is when the device is disconnected from the surrounding Internet.

- The second primary goal is device independent access to data. Mobile computing devices vary dramatically in their processing, storage, and display capabilities. They also vary in the types of operating systems, applications, and user interfaces they employ. UbiData's goal is to make all user data on one device accessible to every other device the user employs. The transferring of data to and from devices is automatic and invisible to the user—eliminating the often-torturous series of steps digital warriors use in today's state of the art.
- The third UbiData goal is propagating users' modifications of data on mobile devices to other mobile and fixed devices through a central data server. The modification of common data by classes of related applications (e.g. word processors) is an oft-occurring event. UbiData adds complexity to this event by propagating only the changes to and from the central data server. Integrating one application's changes to a document into another application's document is one of the most unique features of UbiData.

Use Case

To illustrate our motivations, let us explore a simple scenario that outlines this vision of application- and device-independent synchronization. A busy executive is collaborating on a position paper at his office. His desktop is running a full-fledged word processor, which he is using to write the paper. Prior to leaving for a trip, our synchronization system transmits this document to his PDA. The system automatically strips out pictures, graphs, formatting, and other information that is not usable by the PDA's editing application; it transmits nothing but ASCII text. These transformations speed transmission to the device, minimize storage requirements on the PDA, and still retain enough information for the executive to proofread and make minor modifications to the paper.

During his plane travel, the executive has an important thought pertaining to the position paper. Using the simple text editor on his PDA, he updates the document. When connecting to the company e-mail server in the evening, the PDA's synchronization client also transmits the changes to the document to the central warehouse server for synchronization with the master copy. The amount of data transmitted is a small text-based edit script of a few dozen bytes: minimizing transmission time and facilitating propagation of the changes to other copies of the document. Once the text-based edit script reaches the central server, it gets transformed into intermediate forms of XML and our special XML difference algorithms start their work. This integration of text-only changes back into a full-fledged word processing document is not currently possible with commercial products or research prototype systems.

CHAPTER 2
UBIQUITOUS DATA ACCESS PROJECT

Architectural Overview

The UbiData infrastructure consists of several major pieces. Figure 1 shows a conceptual overview of UbiData. On the right side of the figure are the data sources. Data sources may or may not belong to the users who access the UbiData project's servers. On the left side are the UbiData clients. Each client can publish data to the UbiData server and hence become a data source in addition to being a client. The components in the middle form the UbiData server itself.

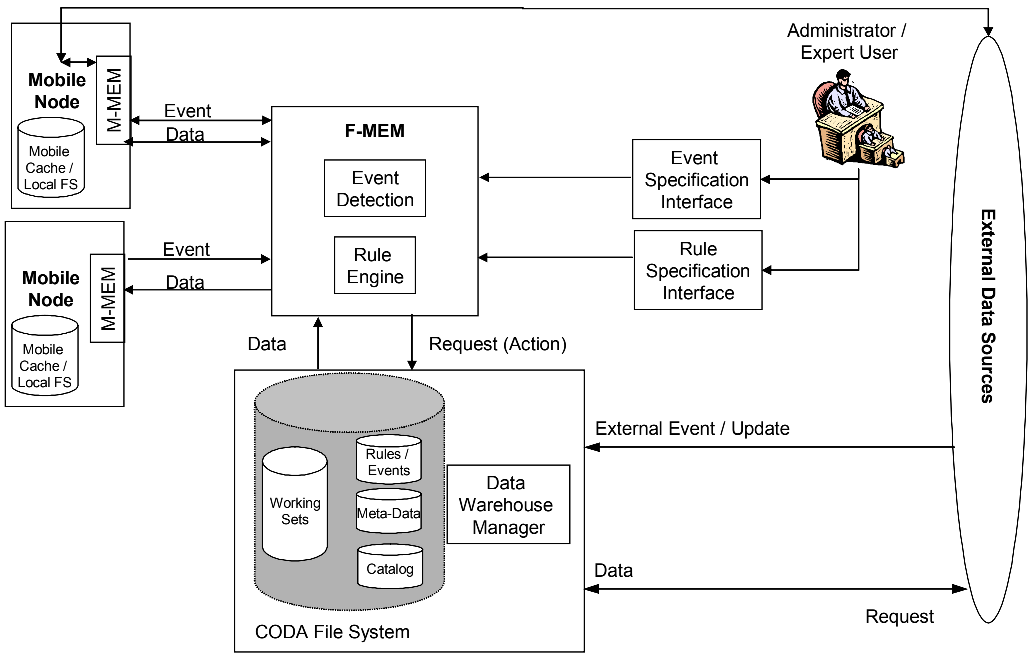

Figure 1 Conceptual Overview of UbiData System Architecture

The system stores not only published data from the sources but also meta-data regarding the contents, type of file, and other critical information. The repository helps maintain consistency levels between the various clients and the data sources using programmable rules. Bringing a client into a consistent state with the UbiData server is automatic and does not require user initiation. Bringing the UbiData server into a consistent state with the data source is also automated: a key factor is whether the data source can send incremental updates to the UbiData server. If the answer is no, our system reverts to value-based hoarding and shipping. Another purpose of the UbiData server is to allow users to set different consistency requirements for their different devices, easing communications requirements on non-critical devices.

The more detailed information depicted in Figure 2 shows the interactions between the UbiData Warehouse Server and the clients.

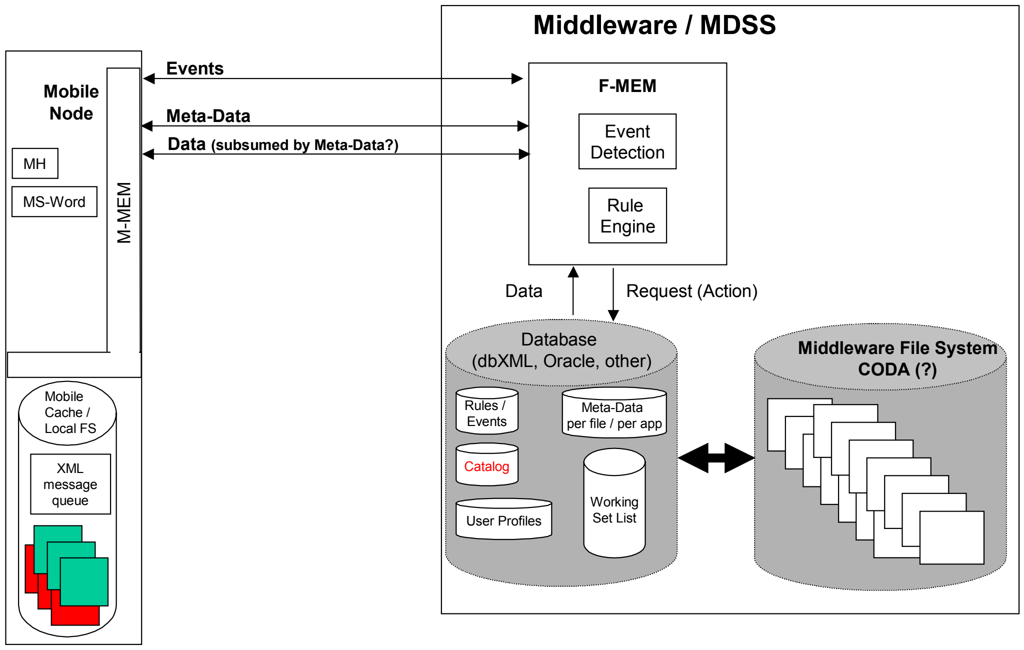

Figure 2 Client Interaction with UbiData Server

We now focus on the synchronization architecture in more detail. The Mobile Environment Manager (MEM) has two primary components: M-MEM and F-MEM (Mobile and Fixed respectively). MEM provides the smart filtering algorithms and communications links between clients and the UbiData Warehouse Server [ZHA01]. Underneath F-MEM on the UbiData server is the meta-data database that tracks the users' working sets, has default rules for device capabilities and filters to employ, and has default conversion rules and tools for moving data between different formats. The database also maintains the links to the actual data files that users have placed in the UbiData server.

The client's M-MEM modules maintain contact with the UbiData server or, upon reconnection to a network, establish connection to UbiData. Along with the initial communication setup, M-MEM identifies its version number, OS type, and device type. M-MEM also transfers the records it has captured during connected and disconnected states: interesting file activities that add to the user's working set, changes in users' desired consistency levels, etc. F-MEM receives these records and proceeds to process them. For files not accessed within a set amount of time, F-MEM and the database drop the file from the working set. For active files, F-MEM receives either entire files or differential updates from the clients. Likewise, when a client first requests (either actively or through an access miss on the client) a file, F-MEM sends the entire file to the mobile device (possibly with intermediate processing).

When M-MEM reports a file access miss on the client, F-MEM must send the file to the client. It must transform the document into a format usable by the client device. Default rules exist for conversion of files being shipped to certain types of devices. The user has the option of specifying additional restrictions on the quantity and type of data to ship: allowing not only a network connection adaptation but also a user patience-threshold adaptation.

Why UbiData Standardized On XML

A principle requirement for cross-application accessibility of data is the ability to transform the original data into a format expected by another application. One way of accomplishing this is to take advantage of the export filters provided by many applications. Using these filters, users may manually export or import a document into formats supported by the application. We wanted UbiData to conduct these conversions automatically and without mandatory user intervention.

We chose Extensible Markup Language (XML) to facilitate these automated and automatic conversions. This choice provides several advantages shown below and in following sections:

- No reliance on proprietary data formats;
- Leveraging the growing influence of XML and using other developer's conversion products for proprietary-format-to-XML and back transformations;
- Leveraging the automation capabilities of Extensible Stylesheet Language (XSL) [WWW01] to standardize transformations of the XML data to various proprietary and non-proprietary formats.

Content Reduction Defined

XML has another advantage when trying to adapt data streams to poor network connections between devices. The anytime, anywhere data access goal of UbiData requires that the server can intelligently adapt to the quality of its Internet connection with the mobile device. UbiData achieves this by transforming XML documents into stripped down versions. It accomplishes this by building an XML document composed of nodes that do not require large amounts of bandwidth and time to download. Elements that surround pictures, graphics, or other large embedded objects simply do not go to the mobile device. Discovering the location of these objects in the document is difficult with proprietary data formats. Since XML is an open-standard encoding scheme, it becomes less difficult and much less likely to violate of emerging legal restrictions on reverse engineering. Figure 3 shows the conceptual view of how the UbiData server incorporates this adaptation scheme into the event flow of communications with a mobile client.

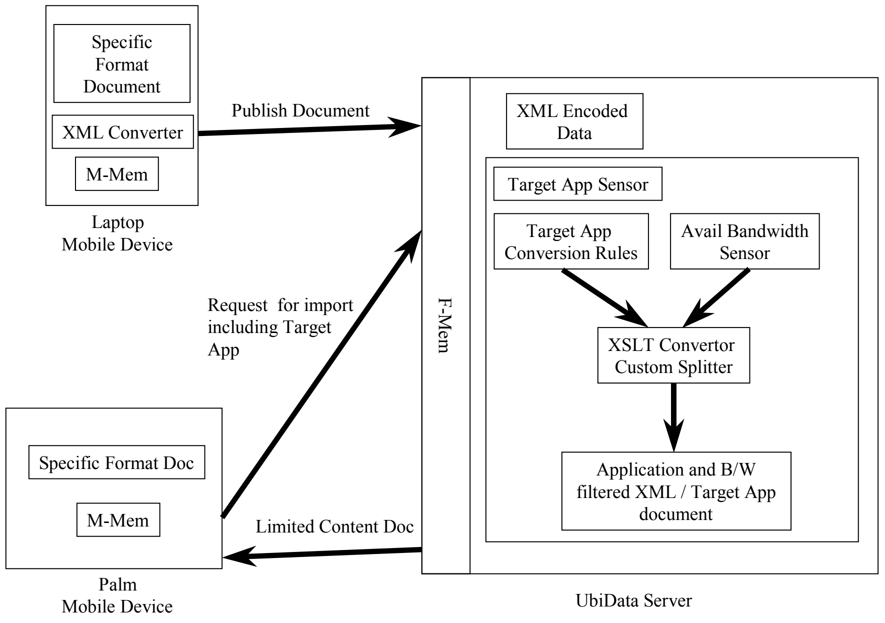

Figure 3 Bandwidth and Target Device/Application Filtering of Source Documents

Difference Algorithms versus Value Shipping

UbiData uses difference algorithms to minimize the amount of data transmitted between the mobile devices and the server. Mobile devices generally have limited battery life. Every transmission of data requires more energy expenditure than transmitting nothing. UbiData wants to minimize the quantity of data transmitted between devices. Such minimization helps extend the life of the battery in mobile devices. It also has the potential benefit of dramatically reducing transmission times.

Difference detection algorithms are not new in the computer science realm. XML difference detection tools are not as mature as older text-based tools, but they are usable and benefit from thirty years of research on difference algorithms in general. A principal advantage to XML difference detection tools over their text-based brethren is XML tools can (but don't always) very efficiently support move semantics. They also take advantage of XML's hierarchical structure. Traditional tools are not even aware of the hierarchal structure of the XML file, so certainly take no advantage of it. Another justification for using XML based change detection tools is that XML documents have no requirement for line breaks between nodes. In OpenOffice's StarWriter, the content document has two lines: the XML declaration line, and the entire document in a many thousand-character line. To capture minor changes inside the XML file, traditional line-break oriented difference detection tools must report the entire second line as changed. This can generate edit scripts with no savings in size with respect to the size of the original document and indeed can by almost twice the size of the original file.

Value shipping offers none of the advantages or capabilities listed above. The simplicity of shipping entire files from one device to another is almost impossible to beat when it comes to implementation. Even consistency checks with value shipping become relatively easy: the user or system picks which of the conflicting files they will keep. Despite these advantages, value shipping demands more transmission time, fails to provide an adaptation capability due to network or patience-threshold considerations, and does not lend itself to space-efficient versioning.

Content Reintegration After Transformations

The final advantage that XML gives the UbiData project is the ability to integrate changes from different applications (different but within the same domain: i.e. word processing) into the original XML document. Current XML change detection techniques can successfully find changes between the stripped down document the server sent to a mobile device and the user's changed document. They cannot then apply that knowledge to the original, non-transformed XML document. To meet the promise of UbiData, our system not only allows reintegration of changed stripped documents, it is built around the requirement.

Our system allows the editing of an AbiWord or StarWriter document in a text editor and having the changes incorporated back into the original XML document. It also allows an impatient user of a laptop that runs StarWriter, the option of omitting pictures and graphics from the data file. With that directive, he receives from the UbiData server the document's contents minus the bandwidth-hungry components that increase download times.

CHAPTER 3
RELATED RESEARCH

The research described here is part of an overall effort to build the infrastructure to support device independent mobile computing. Other device-independent computing efforts are underway at Stanford [AHM95], DARPA [SCH01], IBM [IBM02b], and at Texas A&M [LI02]. The review in this chapter is broken into four major sections that intermesh with UbiData's goals: bandwidth adaptation; traditional text-based difference detection; byte based difference detection; and hierarchically structured (XML) data difference detection.

Bandwidth adaptation is clearly a related topic that encourages and enables different devices to connect to the Internet. In our particular vision, we use static adaptation on the server side of the client/server relationship. Client side adaptation and dynamic adaptation are both providing insights into our approach.

Change detection is fundamental to incremental transmission of data to and from devices. The initial architecture of the UbiData System uses byte based change detection. We have already determined that text-based change detection is not directly helpful, but there are interesting lessons to learn from it. Finally, we use lessons learned by others when attempting to do change detection on hierarchically structured and semi-structured data (like XML).

Bandwidth Adaptation

Puppeteer

The Puppeteer [DEL01a] project at Rice University is closest to our vision of a system that supports ubiquitous computing without modifying the user's applications.

That system focuses on overcoming three obstacles to editing documents on mobile devices. These obstacles are download latencies, the potential for large updates, and update conflicts. To overcome these three obstacles Puppeteer used a combined architecture call CoFi (Consistency and Fidelity) [DEL01b] that supports editing of low-fidelity components of documents.

CoFi, as a sub-component of Puppeteer, supports two classes of fidelity: full and partial. Full fidelity means a data file has all the content created by the original application. All text, embedded pictures, graphics, and other content rich data is present and available for viewing and editing. The second mode is partial fidelity. This mode means a data file has undergone a lossy transformation from full content to a degraded version. The transformations use public Microsoft APIs to parse original MS Office documents. The parsing breaks documents into Common Object Model (COM) [BRO95] and Object Linking and Embedding (OLE) [CHA95] based objects and places those objects into a Document Object Model (DOM). Once in the DOM, their system uses information about the state of the network to transmit the entire tree or just pieces of the tree.

The mobile device's MS Office applications then manipulate the data as usual. Puppeteer also is beginning the process of allowing edits to the low-fidelity components on a mobile device and integrating those changes into the high-fidelity version. Puppeteer does not yet have the cross-application design goal that we are attempting to implement.

Odyssey

Another line of research in bandwidth adaptation is the Odyssey project [NOB97]. Odyssey also adapts application data to the current state of the network connection.

Odyssey has two primary responsibilities to assist it in meeting the demands of application aware bandwidth adaptability. The first is awareness of shared access to remote data. The second responsibility is application and data type specific—the system must have enough information about the data stream to modify it in an advantageous manner. The system fulfills these responsibilities by utilizing three components: *viceroys, wardens,* and *kernel modifications.*

Wardens are components that are specific to each data type. It is a system level component and serves to encapsulate the functionality required to adapt each specific data type. If a data stream consists of motion video, the *warden* contains the methods and functions to adapt the video stream based on input from the *viceroy*. The *viceroy* is a type-independent component that serves as a resource monitor and controller. When resource levels fall outside defined limits, the *viceroy* informs the application. The application then downgrades its expected fidelity or increases its fidelity demands.

Unlike Puppeteer, Odyssey generally requires modifications to applications to support its implementation scheme. The exception is when an application can use proxy servers that Odyssey can then control. This is in contrast to Puppeteer, which uses public APIs of applications to manipulate that application's data files. It is also in contrast to our own system, which utilizes a common format for supported applications: XML.

Alliance

Alliance [DEC95] is a project that focuses on collaborative editing across loosely coupled computing devices. Its design goals are somewhat similar to UbiData. Its intent is to allow multiple versions of a document to exist on many different devices at once. It uses user roles to help manage the views of the data; each user may see some or all parts of any given document depending on their role with respect to that document. Managers see entire documents while writers see the chapters they work on. The similarity with UbiData lies in the presenting different views to the users on different devices.

One primary difference is that Alliance implemented single writer consistency protection. At any given point, of all users who have the potential to be a writer to the central document, only one user can assume the effective role of writer. Such a system does not allow for disconnected writes such as UbiData envisions. Without the ability to write to the document in disconnected mode, Alliance primarily serves as a model for how to deliver different versions of a document to mobile users.

Traditional Difference Algorithms

Simply stated, the purpose of a difference algorithm is to find the differences between two files. It then must represent those changes in as efficient manner as possible. This representation is an edit script. The script describes the edits required of one document to turn it into the other. Each of the edit script's operations (insert and delete) has a relative cost to it. The efficiency of the script is with respect to the sum of those relative costs. The lower the sum, the better the algorithm in terms of finding the least cost edit distance between the original file and the updated file [MEY86].

One of the oldest and most straightforward ways of detecting changes between two files is simple line-by-line iteration through both files. This iteration searches for the first line where the files differ, then searches forward for matches. While the method works, it often represents the differences in a non-optimal way. The edit script incurs costs higher than more intelligent algorithms and is known to be effective, but non-optimal [MEY86].

More recent algorithms such as GNU's *diff()* still derive from work that originated in the 1970s and 1980s. These processes generally performed their operations with complexity of $O(n^2)$ with n the size of the initial input. In [MEY86] the speed of the process was improved from $O(N^2)$ to $O(ND)$, where N is the length of the first input and D is the length of the second input. In the pathological case, the size of the second input can be equal to or greater than the first. In such cases, the complexity reverts to $O(n^2)$.

GNU's *diff()* is capable of treating two files as binary data. The documentation for the program is unclear how it breaks the file into small chunks. It may in fact continue to use end of line character sequences to break the file into lines of characters. An alternative to relying on those characters as the delimiter is to conduct the difference at the byte level.

Conducting change detection at the byte level of granularity is apt to yield better results when dealing with binary data files (as opposed to text data files). The disadvantage of dealing at the byte level for this type of operation is the increased amount of computing cycles needed to perform the operation. At least one tool that conducts binary difference detection uses a more robust way of breaking the files into chunks than *diff()* does.

Binary Difference Algorithms

The Rsync Algorithm and rsync() Program

An algorithm called rsync [TRI96], and a program by the same name, uses a multi-pass scheme to reduce the amount of computations needed by a byte-for-byte comparison algorithm. The algorithm allows change detection of two files on two different machines. A good starting place for such a problem would seem to be ensuring each computer system maintains a copy of the original file as a baseline by which GNU *diff()* can occur. However, *rsync()* starts with the assumption that both computer systems has two sets of data. The user directing the change detection knows there is some relationship between these two sets of data, but is not sure what the changes are.

The algorithm is elegant in its simplicity and accuracy. Given two computers that each have a file related to the other, the algorithm proceeds as follows. For the sake of discussion, we assume the client has a modified version of a document and the server must retrieve an updated copy of the same file. The server proceeds to break its copy of the file into a series of non-overlapping chunks of some fixed size, *S*. It then calculates two checksums against each of these chunks: a 32-bit and a 128-bit MD4 checksum. The server then packages and sends those checksums to the client. The client then searches its file for all blocks of length *S* at any offset from the beginning of the file. For every possible match with the 32-bit checksum, it is confirmed or denied using the 128-bit checksum. The client then sends a sequence of instructions and data back to the server that allows the server to build a new copy of the file.

This algorithm is very good at computing the differences between two related files on separate systems: a goal of UbiData. However, the real similarity between the implementation of *vdiff()* discussed in this thesis and *rsync()* lies in its computation of unique markers for chunks of data. The *rsync()* program cannot support cross application change detection as *vdiff()* does unless each application reads the same type of data file. It also fails to take advantage of any structural information embedded within the file and thus looses the benefits and insights that structure can provide a change detection algorithm. The XML files that this implementation uses are already broken into chunks. The chunks of an XML file are the individual nodes and sub-trees anywhere within the XML document's DOM tree. The *vdiff()* program uses the unique tag lessons of *rsync()*

in developing unique hash values for each node plus unique hash values for each sub-tree.

The *Xdelta()* Program

The Ubiquitous Data Access Project at University of Florida [HEL01] is currently using *Xdelta()* [MAC00] as its means of change detection and versioning for documents under the system's control. The *Xdelta()* File System uses a copy/insert methodology to build its edit scripts. This stands in contrast to standard GNU *diff()* that uses insert/delete operations. A simple example is the insert/delete script for "smart computer" being transformed to "computer smart". This insert/delete script would require six deletes and six inserts. A copy/insert would require three instructions: copy "smart," insert a space, and copy "computer."

The *vdiff()* implementation use some of the lessons generated from *XyDiff()* by not limiting itself to single character inserts and deletes. Though we use insert and delete operations we also support move and update operations. Here a move operation would be equivalent to the copy operation used in *XyDiff()*. UbiData also discovered that *XyDiff()* is not suited to our goal of cross-application difference detection and propagation. Trying to reach the goal of cross-application propagation has forced us to use some common intermediate format for our canonical representation of the data. Those realizations lead us to the utilization of XML encoding of data and XML specific change detection algorithms.

XML Specific Algorithms

Sun Microsystems

SUN Microsystems published a XML change detection tool [WAL00] that utilizes Perl as the implementation language. It uses the longest common subsequence algorithm to find the similar sections of two XML files. The program then proceeds to build an XML edit script that a user may review. The tool provides an XSLT script how-to to allow viewing of the edit script within a browser to see the file differences. The tool does not allow accept/reject options to the changes such as those provided by Microsoft Word's Track Changes Option. The program also assumes the absence of XML data across files is always meaningful (like most change detection methods, it defaults to ignoring white space when executing the difference detection). As far as we can determine, there is no easy method available within Perl to modify its Algorithm::*Diff()* module to ignore certain types of missing XML data.

IBM's XML *diff()* and *merge()* and *xmltreediff()*

IBM has devoted a great deal of resources to expanding usage of Java and XML into its core technologies. As part of that effort they opened and continue to maintain a web site called AlphaWorks [IBM02a]. The site showcases development examples of its programming staff. One of those projects is the XML *Diff()* and *Merge()* tool. It is Java based and utilizes the IBM version of the Xerces-Java XML parser version 1.4. It also supports a GUI to allow a user accept or reject changes between two different XML documents. Unfortunately the source code for this tool is not available for editing and modifications. IBM has also built a Java-beans based *xmltreediff()* [POO99] program that provides both GUI and command-line interfaces. It also has published APIs that allow programmatic access. Unfortunately like *diffmk()*, and XML *Diff()* and *Merge()*, this tool will also fail to meet out requirements in UbiData. We require that the absence of data in a document on a client not be treated as a delete unless the client shares that structure with the document on the server.

The *laDiff()* Utility

The vdiff() tool adopts an implementation method very close to *laDiff()* [CHA96]. However, *laDiff()* does not assume the existence of external identifiers or even unique identifiers to assist in the matching of nodes from one document to another. Without the external IDs, *laDiff()* sacrifices the ability to conduct versioning of the documents. The analytical bound of *laDiff()* shows a $O(ne+e^2)$ with e the weighted edit distance and n equal the number of nodes. However, empirical testing of the routine showed near-linear time with sporadic, but wide variance.

The *XyDiff()* Program Suite

The VERSO Team, from INRIA, Rocquencourt, France wrote this tool for their Xyleme Project [COB02, MAR01]. Their tool executes XML difference detection in near-linear time to the size of the documents. This tool has the original name of *XyDiff()* [COB02] and now has the name *verbose-diff()* (*vdiff()*): accounting for documents' differing levels of structural verbosity and content. *XyDiff()* expanded on the capabilities of other XML tools by incorporating the ability to capture move and update semantics. Like *laDiff()*, *XyDiff()* is one of the few XML tools to utilize the move semantic for edit scripts. This takes advantage of the hierarchical nature of XML and allows movements of an entire sub-tree to new locations with a single entry in a edit script.

The original *XyDiff()* algorithm utilizes external identifiers (Xyleme IDs)[MAR01] to permanently identify each node in the original (*v0)* XML document. These identifiers correlate to a post-order traversal of the DOM tree created by parsing the XML document. This post-order traversal follows the convention used by [SHA89] in tree-to-tree correction problems. The XIDs are crucial to the versioning capabilities of *XyDiff()*

and to our modifications. The remainder of their algorithm is shown below and worth noting prior to exploring the *vdiff()* algorithm.

```
v0DOM = ParseAndLabel (v0document, isSource);
v1DOM = ParseAndLabel (v1document, isNotSource);
BuildSubTreeLookupTable (v0DOM);
FindAndUseIDAttrs (v0DOM);
TopDownMatchHeaviestTrees (v1DOM);
PeepHoleOptimization (v0DOM); //force matches if reasonably safe
MarkOldTree (v0DOM);
MarkNewTree (v1DOM);
BuildLeastCostEditScriptForWeakMoves (v0DOM, v1DOM);
DetectUpdatedNodes (v1DOM, v0DOM);
ConductAttributeOperations (v1DOM, v0DOM);
WriteXIDmapFile (v1DOM);
WriteDiffInfoToFile ();
```

Figure 4 Pseudo-Code Algorithm for *XyDiff()*

The VERSO team discovered through empirical testing that this algorithm operates at near linear speed on documents up to 10 megabytes in size [COB02]. Further discussion of the algorithm remains for a later section. The INRIA team, specifically Gregory Cobéna, has provided numerous insights into their code to assist in the modifications that formed *vdiff()*.

CHAPTER 4
ALGORITHM ANALYSIS AND DESIGN

Use Case (Revisited)

The mobile client in the use case is a personal digital assistant (PDA) that is not capable of rendering complex documents and their content. Our executive has been working on an AbiWord generated document on a device not pictured below in Figure 5. The server has placed the document into the user's working set and has a copy stored on its own storage device. When the PDA connects to the server, the server updates the PDA with the contents of files not previously in the working set and therefore not on the PDA. In our case, the server initiates an automatic conversion from AbiWord's XML format to a text-only format suited to the small storage and display capacities of the PDA. It then transmits the data to the client via communications handled by F-MEM and M-MEM.

The executive re-reads his document on the PDA for proofreading purposes. He makes several spelling corrections, grammar corrections, and repositions several paragraphs. When he finishes, M-MEM uses GNU *diff()* to compute the changes to the document. M-MEM then packages those changes and awaits another opportunity to transfer the data to F-MEM.

When F-MEM receives the message with the edit script embedded in it, the server then applies the edit to the text-only document it kept. The work in this thesis did not integrate M-MEM and F-MEM with *GNU diff()* and *GNU patch()*. Manual execution of those commands is the current simulation technique we employ. We now delve into a deeper explanation of each of the intermediate steps we take to reach own end-state: an XML encoded edit script describing the changes the user made on the palm that can be incorporated into the original document.

The conceptual overview of the algorithm needed to implement our vision of UbiData, as needed by our use case, is show in Figure 5. It is important to note that for mobile devices with too little computing power to support *vdiff()*, the system requires two iterations of change detection: once with GNU *diff()* on the mobile client and once with *vdiff()*on the server. Figure 3 shows the sequence of operations for a mobile device that can support *vdiff()*. Only one round of change detection occurs: on the client. The client then ships the edit script back to the UbiData server for propagation to the master copy of the document.

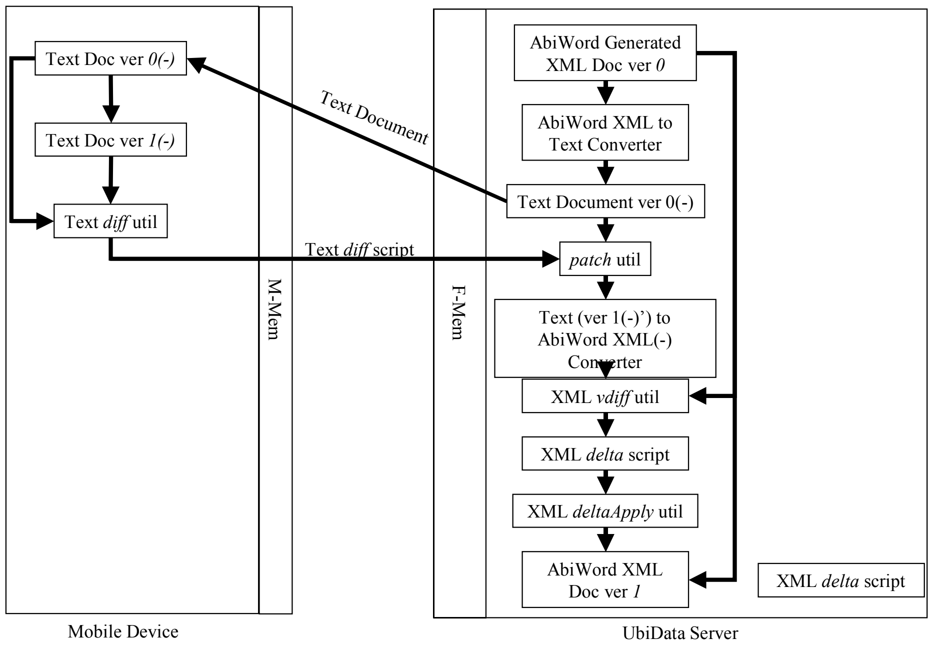

Figure 5 Use Case Overview with a Text-Only Mobile Device

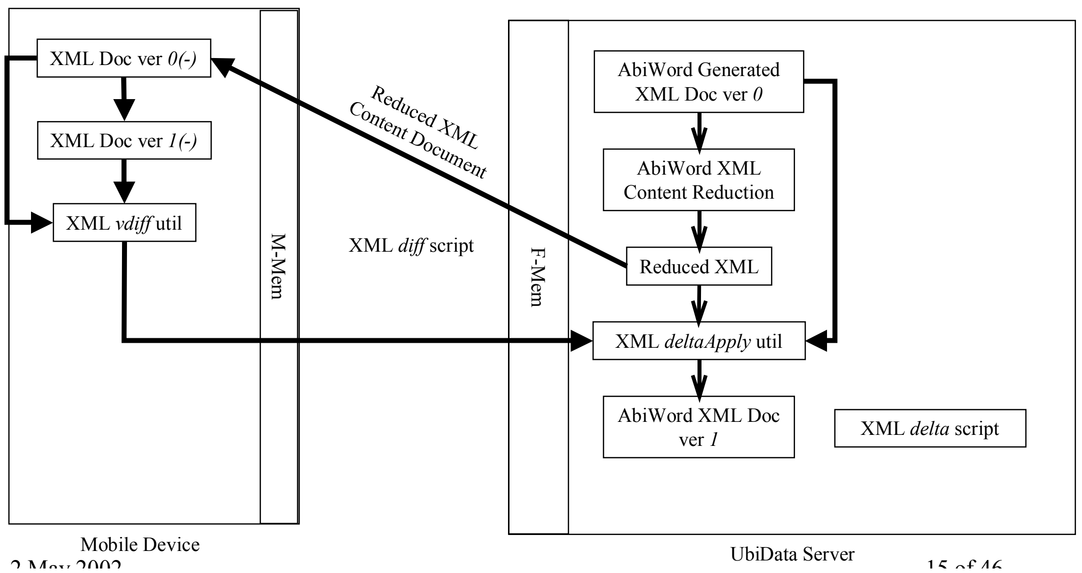

Figure 6 Use Case Overview With a Computationally Powerful Mobile Device

The information in Table 1 helps clarify the various version numbers in Figure 5 and Figure 6.

Table 1 Document Version and Description Legend

Version	**Description**	**Version**	**Description**
v0	Rich Content, XML Doc	v1(-)	Changed version of v0(-)
v0(-)	Reduced Content and/or transformed version of v0	v1(-)’	If required, v1(-) with XML structure imposed
		v1	V0 with modifications from v1(-)

Content Customization

Content Reduction

Content reduction is the process by which the UbiData System will deliberately remove bandwidth hungry objects from a data stream. The automated functionality allows for the system to use intelligent defaults when faced with less-than-optimal connectivity to the requesting computing device. The other benefit of having this feature is the user does not need to know precisely what elements of a document his device’s applications can use. Additionally, when the user overrides the system and specifies removal when none would normally occur, the user chooses to save time over data content.

Both AbiWord and OpenOffice use XML as their data encoding method. This makes the task, if not trivial, a matter of pure mechanics instead of theoretical science. This capability is not yet implemented in UbiData. No predictions exist on the ease of building the mechanisms to strip bandwidth hungry objects from these two, and other, XML data streams. After all, repairing a lawnmower's engine may not be theoretically difficult, but many a shade tree mechanic has ruined their mower with under estimations of the skills required. Puppeteer has proven this content reduction is feasible and executable in MS Office Documents [DEL01a]. It now remains a matter for execution within the realm of OpenOffice and AbiWord as test document generators.

After such a grim introduction, let us probe the specifics of the tasks for content reduction. Analysis of AbiWord documents and OpenOffice documents show they both use specialized tags to hold both meta-data and the actual content of objects such as pictures, OLE objects, and graphics. The tags for both to encode the presence of a picture is <image>, and of tables <table>. When the XSL-driven or custom-converter encounter's such tags, the tag gets skipped, replaced with a placeholder, or uses another technique to help reduce the transmission time. This is a static adaptation to the network connection between the UbiData Server and the mobile device. There are currently no plans to dynamically change the quantity of data shipped to a client in the event a network connection dramatically improves during transmission.

AbiWord embeds pictures' actual binary data in the overall document. In OpenOffice, the content file's <image> tag has an href attribute that points the application to another file. At this point it is worth a small digression to discuss the content structure of an OpenOffice file.

OpenOffice uses XML as its encoding for all documents that it creates: word processing, spreadsheet, and presentation. It stores the content into a zip archive with the name provided by the user. A typical zip archive expands to look like that shown in the.

Table 2 Contents of an OpenOffice Document Zip Archive

File Name	Description
Pictures/07314EC41FE.wmf	Embedded picture in native format
Pictures/0B21A10A201.jpg	Embedded picture in native format
Pictures/07369586D1F.wmf	Embedded picture in native format
layout-cache	Binary file
styles.xml	Text formatting styles
Settings.xml	Application settings at time of save
Content.xml	Text content of document, spreadsheet, etc.
meta.xml	Document author name, other meta data
META-INF/manifest.xml	Listing of expected contents of zip archive

XML <image> tags use attributes specify whether to activate the link upon load or other times. Such cleanly delineated markers will greatly simplify the removal of these objects from the transmission stream. Those markers stand in stark contrast to locating data with a proprietary data format. Cleanly removing content from data files created by OLE or COM enabled applications becomes significantly more of a challenge in a heterogeneous environment.

Content Conversion

This implementation uses custom C++ code and open source software to create the first generation converters for the UbiData system. Conversion of the XML documents created by AbiWord and OpenOffice to ASCII format text files proved a matter of mechanics. As the complexity of the XML documents involved with the test cases rises, it is possible that the current tools will need modification.

Our conversion utilized the Xerces-C++ XML parser from Apache.org [APA02b] plus custom C++ code to isolate and print appropriate nodes. Future conversions may continue using this approach, but work is proceeding on utilizing XSL and the Xalan XSLT [APA02a] processor (again from Apache.org). It is apparent that UbiData will have to have an XSL style sheet or custom converter for every application we intend to support. One may argue that it is simpler to utilize an application's built-in export feature often suffices for the transformation to other formats—no built-in converters we are aware of is capable of omitting selected structures of the document. Such omission is what allows UbiData to adapt to network connectivity status or user patience.

The conversion process generates text that not attempt to recreate positioning of text (i.e. centered, right margined, justified) nor does it contain any formatting other than new lines, form feeds (when applicable), and tabs. This first step, and further conversions and manipulations of test documents are shown in Figure 5.

Meta-Data to Support Content Customization and Structural Similarities

We discuss the concept and actual implementation of content customization. What we have not discussed so far is if or how we would track this customization. Indeed it is not a question of if we track the data. Without knowledge of what UbiData does not include in its document transformation, the system has little hope of being able to integrate a user's changes back into the original document. In the following sections we use the terms *v0* document and *v1* document to mean the original document served by the UbiData and the modified document residing on the mobile client.

General

The *v0* document contains a significant amount of data and meta-data never converted to text and utterly useless to a text editor (hence the reason it was not converted in the first place). This data, in the case of AbiWord and OpenOffice, contains style information, page definitions, dictionary data, and other information the application embeds in the data file and data archive. Other XML change detection tools [IBM02a, CHA96, WAL00] will interpret the absence this data in the modified document (*v1*) as deletion of the corresponding nodes. If those delete actions propagate to the *v0* document, it is immediately apparent that the *v0* document will become corrupted and possibly unusable to the application.

Tracking what structures two XML documents have in common, or alternatively do not have in common, is fundamental to *vdiff()*'s approach. Without this information, no roundtrip is possible between a transformed and edited document back to the originating document. To accomplish this tracking we rely on piece of externally stored data. We created a XML schema that defines an intersection or symmetric difference map file. The map lists the originating application of the *v0* document (e.g. AbiWord) and the target format of the transformed *v1* document (e.g. text).

The schema then allows for a tag-by-tag enumeration of shared and aliased tags or a tag-by-tag enumeration of unshared tags and unshared attributes. The ability to list only the intersection of identical tags between XML formats prevents having to enumerate every possible tag name in the domain of the *v0* document. The alternative is to list only the unshared nodes (the symmetric difference) between two XML documents. Figure 7 shows the *vdiff()* schema, as generated via XMLSpy with the intersection element expanded while Fig and Fig graphically represent overlapping tag sets and symmetric difference tag sets.

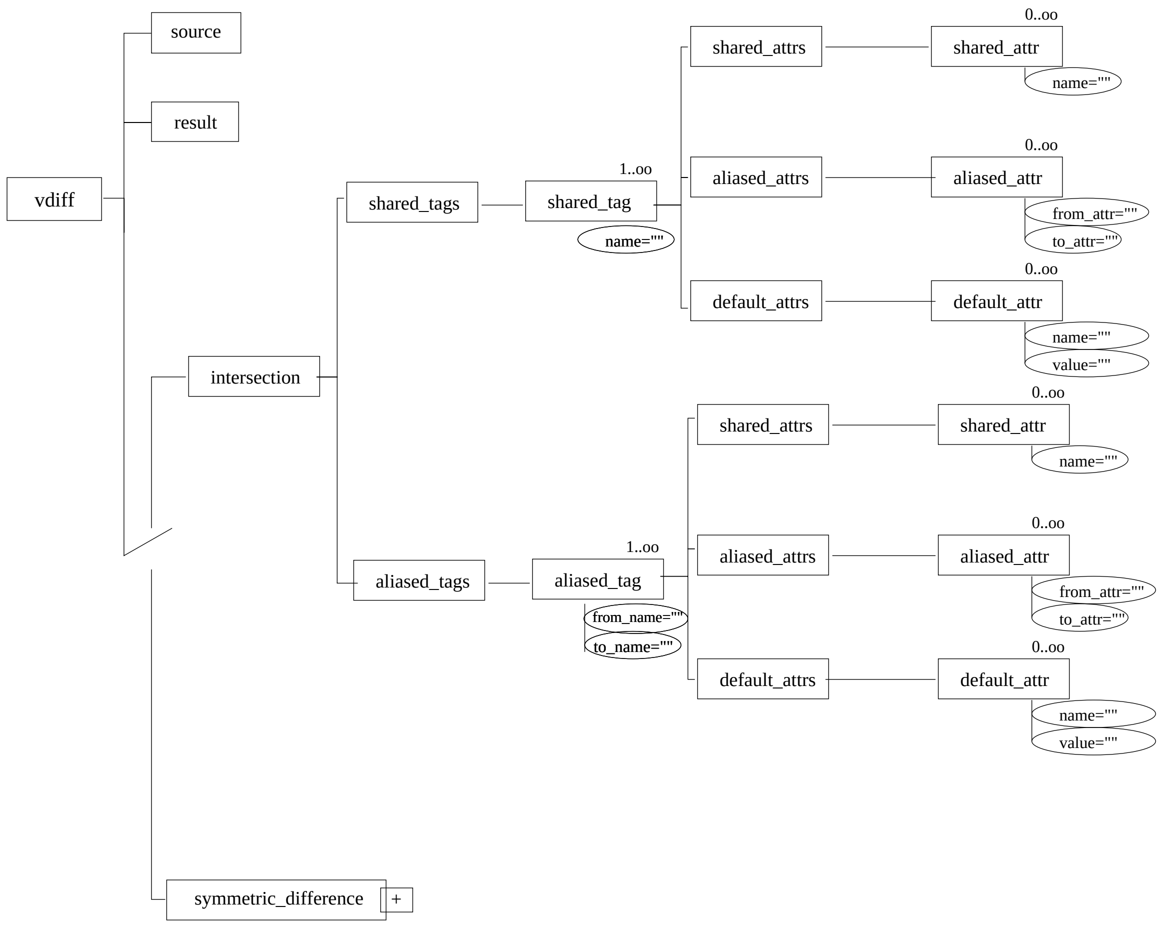

Figure 7   Intersection and Symmetric Difference Schema for *vdiff()*

Intersection Map

The intersection of tag names between the v0 and v1 documents is essential knowledge. In the absence of this map file, the algorithm assumes that both documents have a complete overlap in allowed element tags. As such, the program treats all data within the XML files as meaningful. When this map file exists, the program gets to perform extra processing to determine if absence of data in v1 corresponds to intentional delete actions by the user.

It is worth noting the existence of the <default_attr> child element of both the <shared_tag> and the <aliased_tag>. The capability to embed default values for tags becomes essential in the test cases we cover later. When we detect changes between the minimalist XML structure (*v1* document) against the *v0* document, the <p>aragraph tags do not have attributes. Eventually the program determines some <p> tags are insert actions, and it uses the default attribute values when inserting the <p> tag into the edit script. If this inference of default attribute values did not happen, AbiWord will not function correctly. More accurately, the program will not render a paragraph without attributes. To the AbiWord user, it will appear that the text he inserted on the PDA did not make it back to his desktop AbiWord. In actual fact, the paragraph and its child nodes do exist in the document: the application simply does not render then in a visible manner.

We provide the infrastructure to list aliased tags for future expansion of the system. The most likely scenario for this expansion is transforming XML documents on the UbiData server into HTML. The intent would be to allow the users of the mobile devices to edit the HTML file with no knowledge that it derives from structured XML. Since HTML has a more limited set of possible tag names, it is likely that multiple XML tags will map to a single HTML tag. Knowing this information will assist reconstruction of the XML document from the modified HTML file.

In Figure 7 we can see that our intersection map will include the shared tag labeled 3 and n. The v*diff()* generated scripts will not delete from the *v0* document any tags that do not have these two labels.

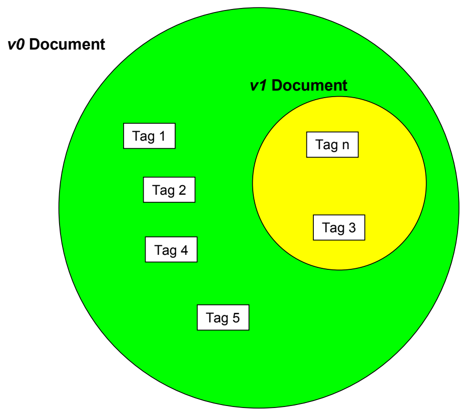

Figure 8 Intersection Map

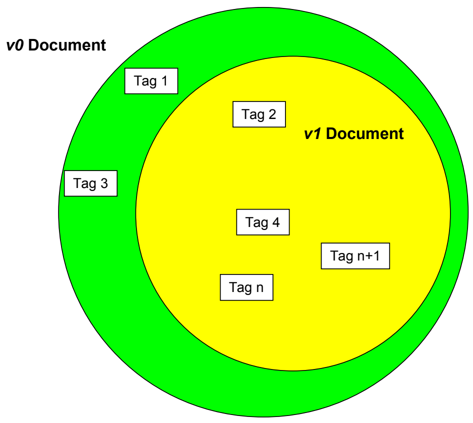

Figure 9 Symmetric Difference Map

Symmetric Difference Map

The Symmetric difference is easily identified by the enumeration of the tag names the two documents do not have in common. This can save an enormous amount of work when both documents have a very large tag set. In Figure 9 we can see that the Symmetric Difference Map will include only the tags labeled 3 and 1. *Vdiff()* generated scripts will not delete any tags from *v0* that have the labels of 1 or 3. It is also possible, though no example is readily available, for two documents to share a common tag name, but not all of that tag's attributes. This is why the symmetric difference allows for listing unshared attributes. Of necessity, the two documents must share the element name and then enumerate all the unshared attributes.

The reader should note that the *v1* document tag cannot exceed the boundaries of the *v0* tag set. If this happened, then the changes sought by *v1* would definitely render the patched *v0* document invalid against its DTD or schema.

Client-Side Difference Detection

As shown in Figure 5, the current version (*n*) of a text-only document propagates to the mobile device by a user-specified hoard directive. The file may also get sent to the

PDA automatically due to its presence in the user's working set. Once the document is on the mobile device the user can read and write the data, making changes as needed. Upon a file-close event, the M-MEM module uses an incarnation of GNU's *diff()* [FSF02b] to process the differences between the downloaded text file and the modified text file. M-MEM then packages this edit script and sends it to F-MEM. The integration of M-MEM with GNU *diff()* is not currently in-place for PDA devices, but on laptop's the integration is complete using Xdelta `[MAC00]`. For now we simply simulate the integration by calling *diff()* ourselves.

F-MEM applies this edit script to the copy of the text-only file originally sent to the client. This automated patching is currently simulated by direct intervention. The GNU *patch()* utility modifies the text document remaining on the server, converting it to version *n+1*.

Imposing a Minimal XML Structure on Text Documents

A change detection algorithm between unstructured text and a structured XML document is extremely unlikely to produce any meaningful information. It is apparent that the system must express the modified *n+1* document in XML format. It is also apparent that it is impossible to impose a full-fledged structure upon the text-only document—we have no information by which to judge what sections of the text belong to what structures in a verbose XML document. At this point, a special tool created using *flex* [FSF02a] and C++ imposes a minimal XML structure upon the text document. This imposition is specific to the class of XML document we are trying to compare the text to. This minimal structure is a result of analysis of AbiWord and OpenOffice documents and has minimal tag sets that it employs.

Briefly, the flex-generated tool starts the creation of an XML file with the XML declaration, and the first two elements: <abiword><section>. It uses several simple states to determine when to close and reopen <section> tags, open and close <p><c> </c></p> tag pairs for paragraphs, <c> </c> tags for tabs, and<p/> tags for empty lines. This example is highly specific to AbiWord. The example also shows how this system will require customization for each XML-to-specified-format transformation we want to support. Similarly, we need, and have, a converter to impose a primitive OpenOffice structure upon text documents. The primitive OpenOffice tag set is even smaller with only <document_content>, <text:p>, <text:s> currently in use. Conversion of special characters like &, “, ‘, < and > into XML entities also happens during this imposition of structure.

XML Difference Detection—*verbose-diff (vdiff())*

We continue to use the XIDs (renamed eXternal IDs [aka XIDs]) of the *XyDiff()* tool to provide permanent identifiers for every node in the *v0* document. The pseudo-code below in Figure 10 provides the top-most level of the *vdiff()* algorithm and assumes three arguments are provided by the invocation, the name of the *v0* document, the name of the *v1* document, and the name of the map file describing structural similarities or differences. In the code, the v0 document is the originating document (usually full-content and full-structure XML). The v1 document is the modified document (in our use case the modified text that had a primitive XML structure imposed on it by *vdiff()*)

```
1.     v0DOM = ParseAndLabel (v0document, isSource);
2.     v1DOM = ParseAndLabel (v1document, isNotSource);
3.     StructuralMapInfo = ParseMapFile (mapFile);
4.     BuildSubTreeLookupTable (v0DOM);
5.     FindAndUseIDAttrs (v0DOM);
6.     TopDownMatchHeaviestTrees (v1DOM);
7.     PeepHoleOptimization (v0DOM);
8.     MarkOldTree (v0DOM, StructuralMapInfo);
9.     MarkNewTree (v1DOM, StructuralMapInfo);
10.    AdjustForUnSharedChildren (v0DOM, v1DOM,StructuralMapInfo);
11.    BuildLeastCostEditScriptForWeakMoves (v0DOM, v1DOM);
12.    DetectUpdatedNodes (v1DOM, v0DOM);
13.    ConductAttributeOperations (v1DOM, v0DOM,
                                  StructuralMapInfo);
14.    WriteXIDmapFile (v1DOM);
15.    WriteDiffInfoToFile ();
```

Figure 10 Pseudo-Code Algorithm for *vdiff()*

The `ParseAndLabel` method of line 1 and line 2, uses the Xerces [APA02b] XML parser to parse and validate the input XML files. Immediately after parsing, the method then traverses the in memory DOM tree and builds a mapping between each node and its XID.

Line 3's `ParseMapFile` method simply processes the contents of the file and instantiates the appropriate classes and data structures. The data structures are primarily STL maps that ensure *O(1)* lookup. This is essential to ensure little performance penalty when checking to see if an element tag is shared between the two documents.

The `BuildSubTreeLookupTable` method on line 4 traverses the v0DOM-tree and builds an average case *O(1)* lookup table of sub-trees. The key for each sub-tree is a hash value created from the content of the sub-tree's root plus the cumulative hash values of its children.

The method on line 5, `FindAndUseIDAttrs`, uses the DTD of the source document, if present, to determine if any elements can have ID attributes. If such elements can exist in the document, traverse the v0DOM-tree and v1DOM-tree attempting to find the appropriate matched nodes. Because XML ID attributes must be unique identifiers, no further processing is necessary to make the match between a node in the v0DOM and the v1DOM. The identical value in the two ID attributes is *prima facie* evidence of a match.

`TopDownMatchHeaviestTrees,` line 6, uses the sub-tree lookup table built at line 4, to search for matches starting at the top of the v1DOM-tree. An obvious and mandatory match of the root nodes happens first, then a breadth-first search. If a match occurs at a non-leaf node, then recursively assign all the descendants of the matched nodes.

Line 7 shows the next step, `PeepHoleOptimization`. This is an attempt to increase the number of matched nodes without incurring inordinate risks of creating false matches. Briefly, if two nodes are matched, build a list of their respective unique unmatched children. For every unique element tag in that child list, if there is only one instance of that tag in each child list, match the child nodes. If there is more than one instance of the tag, there is insufficient data to force a match.

Lines 8 & 9,`MarkOldTree` and `MarkNewTree,` are relatively straightforward. They traverse the v0DOM-tree and mark as deleted every node not matched and whose tag both documents share. The determination if the documents share a tag is through the *O(1)* lookup tables built inline 3. Also mark nodes strong moved if they and their matched node do not have parents that are themselves matched to each other. For the v1DOM-tree, mark unmatched nodes as inserted.

`AdjustForUnSharedChildren`, line 10, is a key component to ensuring the proper order of inserted and moved children. Without compensating for the offsets caused by unshared children, a node's child list will not be in a correct sequence. For example, an inserted child may appear to be the *i*th child of a node in the v1DOM-tree. When incorporated back into the original document however, it should rightly be in the *j*th position. If left with an incorrect insertion position, the edit script will insert the node at the *i*th position. This incorrect positioning will cause errors as minor as wrongly ordered paragraph/picture sequences. The incorrect insertion point can also cause errors as major as violating the DTD or Schema of the source document.

The `BuildLeastCostEditScriptForWeakMoves` method of line 11 is a straightforward longest common sub-sequence problem. The task is to determine the least expensive means by which each node can turn its old child sequences into its new child sequence. The cost for inserts and deletes are proportional to the weight of the sub-tree each child represents.

Line 12 has the `DetectUpdatedNodes` module. If a node and its matched node have only singular unmatched text-node children, match the text-nodes. Consider the text-nodes updated and assign new XIDs to them. While straightforward in, testing revealed this inherited code from *XyDiff()* is fundamentally broken. The specifics of how this module does not meet its design intentions are left for the results section of the thesis.

In line 13, `ConductAttributeOperations` looks at every matched node and determines if its attributes represent inserted, deleted, or updated values. We again use the StructuralMapInfo, built in line 4, and its *O(1)* lookups to minimize lookup expenses. The lookup determines if the absence of attributes in a v1DOM-tree node are meaningful. In the domain of our test sets, this function also infers attributes for inserted nodes. This inference is critical to the proper rendering of paragraphs in both AbiWord and OpenOffice. Attributes contain the style, font, and other formatting information for every paragraph node.

More complete details of the implemented algorithm and discussion of implementation hurdles are left for the next chapter. The current implementation also has significant areas where future research can improve the performance significantly.

XML Difference Application/Patching

Applying the *vdiff()* generated edit script is an uncomplicated task. It consists of parsing the v0 document that remains on the server and the edit script itself. The patch program reads the externally stored XID file to correctly number each node in the v0 document while conducting its post order traversal. The patch program then reads the edit script and follows the sequence of instructions for each node in the script. This tool remains unchanged from that provided by INRIA's VERSO team.

CHAPTER 5
IMPLEMENTATION AND IMPLEMENTATION HURDLES

Content Reduction

The first generation tool that converts our AbiWord documents to text uses a fairly simple process. The Xerces XML parser reads, parses, and validates (as needed) the input file. We took advantage of it's built in transcoding mechanism to print all <p> and <c> tags' text-nodes to the output file. This is admittedly a simple process and not really very interesting.

The second-generation tool will include the printing of stylized text to serve as placeholders for data that cannot be easily rendered as text or would loose too much information if translated into nothing but text (e.g. tables). On-going analysis of AbiWord and OpenOffice files will provide further information on what structures can be reasonably shown as raw text.

Client-Side Difference Detection

One of UbiData's goals is to minimize the work a mobile device needs to accomplish to access and change data that originated on other devices. We anticipate, but have not implemented, a GNU *diff()* utility on the PDAs that will access UbiData. For the purposes of this research, we have simply used the test bed computers to simulate the receiving PDA. We also simulate the integration of the *diff()* utility with M-MEM by manually starting the program and manually applying the edit script to the text document that will reside on the UbiData server.

Imposing a Minimal XML Structure on Text Documents

The subscribers of USENET news groups that focus on XML are a broad based and knowledgeable group. Suppose we took a poll and asked them, "Is it possible to convert a text-only term paper into XML?" It is likely that the vast majority would flat say "No." Dissenters would qualify the "No" by placing a large group of pre-requisites in front of their answer. While this research concurs there is no generic solution to imposing a given XML structure upon a text only document, we approach the problem in a less complicated manner.

Our implementation does not need to impose the complete AbiWord document structure upon an edited text file. Nor would this be possible with large, and likely incorrect assumptions about styling, language encoding, and any number of other variables. Instead, our implementation uses non-automated analysis of the AbiWord file format to discover that all paragraphs and text reside as children to either <p> nodes or <c> nodes. Over 90% of the nodes in our test cases had all paragraph text as a text-node child of the <c> tag. This pattern caused paragraphs to look like the following pattern: <p *style attributes* > <c *more style attributes*> paragraphs or other text </c></p>.

This allowed use to realize we can impose a minimalist XML on the text document by using a small sub-set of the node names AbiWord considers valid: <abiword>; <section>; <p>; and <c>.

Figure 11 below shows the states we programmed into a simple lexer built with the help of flex. This conversion from text to XML has similarities to what future efforts will face when converting HTML or other formats to XML.

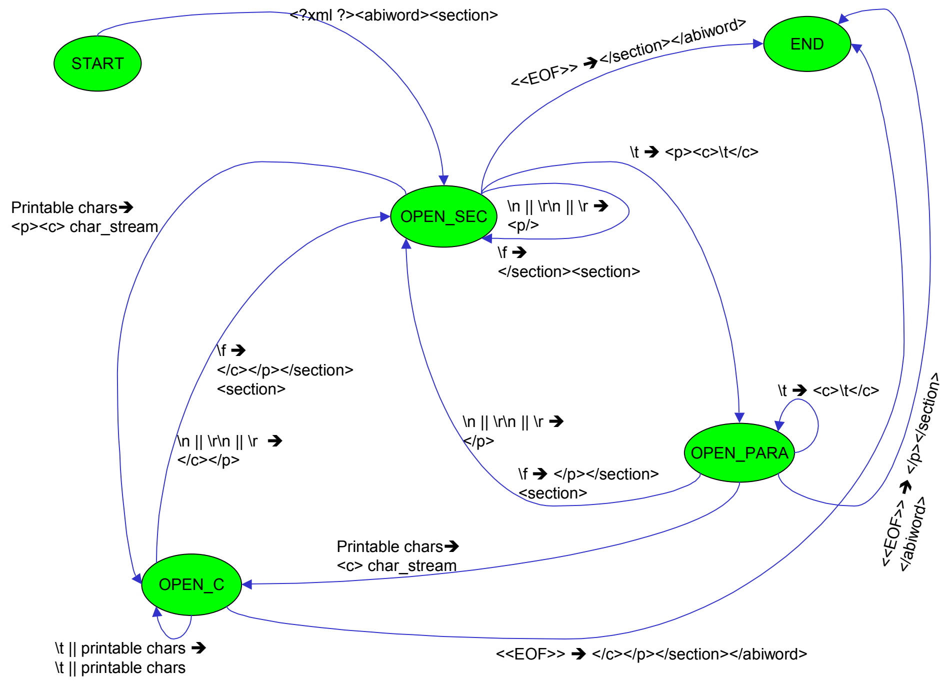

Figure 11 State Transition Diagram for Text to AbiWord Converter

The input character stream drives the movement from one state to another and triggers writing XML tags to the output stream. The tags enclose the triggering input characters except in the face of the end of line characters and form feed character. We chose to write the XML file as a two-line file: the XML declaration line and a second line with all the nodes. This file is not easily read by humans, but is very efficient at preventing false matches of end of line characters in the v0 document.

Algorithm Review

It is worth taking a moment to revisit the short version of the algorithm introduced in the previous chapter. From this point forward, the discussion will delve into the interesting details of the *vdiff()* program.

```
v0DOM = ParseAndLabel (v0document, isSource);
v1DOM = ParseAndLabel (v1document, isNotSource);
StructuralMapInfo = ParseMapFile (mapFile);
BuildSubTreeLookupTable (v0DOM);
FindAndUseIDAttrs (v0DOM);
TopDownMatchHeaviestTrees (v1DOM);
PeepHoleOptimization (v0DOM);
MarkOldTree (v0DOM, StructuralMapInfo);
MarkNewTree (v1DOM, StructuralMapInfo);
AdjustForUnSharedChildren (v0DOM, v1DOM, StructuralMapInfo);
BuildLeastCostEditScriptForWeakMoves (v0DOM, v1DOM);
DetectUpdatedNodes (v1DOM, v0DOM);
ConductAttributeOperations (v1DOM, v0DOM, StructuralMapInfo);
WriteXIDmapFile (v1DOM);
WriteDiffInfoToFile ();
```

Figure 12 Pseudo-Code Algorithm for *vdiff()*(Revisited)

The *vdiff()* processing occurs in a series of traversals of the XML DOMs representing the two documents. As shown above, the first bottom up pass is to identify all nodes that have ID attributes. The next pass is top-down and attempts to match sub-trees to each other: the bigger the trees the more content is unchanged between documents. The final matching step is what we have referred to as peephole optimization.

Component Implementations

ParseAndLabel

The `ParseAndLabel` portion of the algorithm is very straightforward. The Xerces parser is very easy to use and Apache.org has provided superb documentation for the API. After the parser completes the reading and processing of the file, we proceed to use label the nodes with XIDs. If an existing XID map file exists, parse and use it to ensure accurate versioning. If no XID map file exists, it is the first time *vdiff()* has encountered the file. *vdiff()* conducts the post order traversal inserting each XID and DOMNode pair into two STL maps: XIDbyNode and NodebyXID. This method is unaltered from the *XyDiff()* tool.

ParseMapFile

The `ParseMapFile` method simply processes the contents of the file and instantiates the appropriate classes and data structures. The data structures are primarily STL maps that ensure *O(1)* lookup. A complete review of every class and supporting method will add little to this discussion. Implementation of this task was not difficult and the justifications for the structures implemented are in the previous chapter.

BuildSubTreeLookupTable

`BuildSubTreeLookupTable` traverses the v0DOM and builds an average case O(1) lookup table of sub-trees. It does this but building a vector of maps. The maps use tree hash values as keys, and vectors of XIDs as the data member. Each position in the outermost vector is the number of generations between any given node and the root. The map at each vector index stores the hash values of every node between a given node and that parent. This is also unaltered from *XyDiff()*.

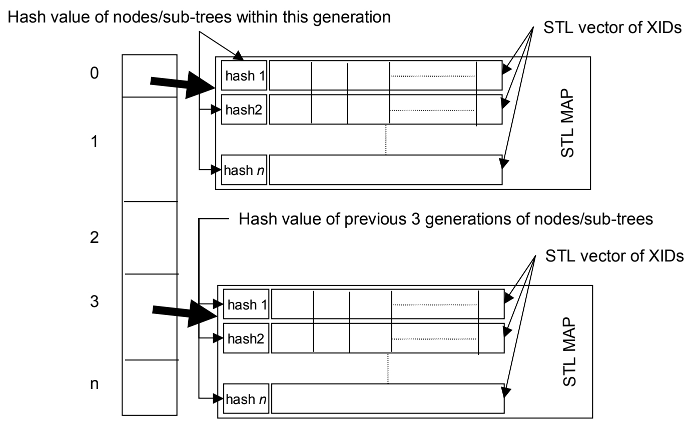
Hash value of nodes/sub-trees within this generation
STL vector of XIDs
0
1
2
3
n
hash 1
hash2
hash *n*
STL MAP
Hash value of previous 3 generations of nodes/sub-trees
STL vector of XIDs
hash 1
hash2
hash *n*
STL MAP

Figure 13 Sub-tree and Hash Lookup Table used for Top-Down Matching

FindAndUseIDAttrs

`FindAndUseIDAttrs` uses the DTD of the source document, if present, to determine if any elements can have ID attributes. Out implementation also does not take advantage of

Xerces' XML Schema support. This is a shortcoming that will limit the useful life of *vdiff()*. No change occurred between *XyDiff()* and *vdiff()* yet, but *vdiff()* will provide schema support soon.

TopDownMatchHeaviestTrees

`TopDownMatchHeaviestTrees` uses the sub-tree lookup table built to search for matches starting at the top of the v1DOM. An obvious and mandatory match of the root nodes happens first, then a breadth-first search. During this breadth first search, each child node of the root (and its children) checks it's own and its sub-tree hash value for a match starting at the one (1) index of the lookup table. If it does not find a match, then it uses the map at index two (2) of the lookup table. This continues up to a fixed number of generations or until the algorithm passes the root. If either happens, it then conducts a search of the map at position zero (0). Obviously, the further up a node travels on its ancestral path towards the root, the bigger the sub-tree being matched. Equally plain is that few big sub-tree matches is preferable to hundreds of small single node matches. If a match occurs at a non-leaf node, then recursively assign all the descendants of the matched nodes. Though the data structure that supports this methods is somewhat convoluted, it remains as it is in *XyDiff()*.

PeepHoleOptimization

`PeepHoleOptimization` occurs next and is the last attempt at matching nodes in this implementation. Due to its importance, we discuss this step in the next section.

MarkOldTree and MarkNewTree

`MarkOldTree` and `MarkNewTree` are relatively straightforward. Adjustments to these methods from the INRIA algorithm include utilizing the Structural map file discussed in section entitled Meta-Data to Support Content Customization and Structural Similarities.

Traverse the old tree and for every node not matched, if the two documents have this tag name in common (*O(1)* lookup using the StructuralMapInfo) mark it deleted.

Mark a node strong moved if it and its matched node do not have parents that are themselves shared and matched to each other. When the v0DOM is not as flat as it is in AbiWord the determination of a strong move is more complicated than at first glance.

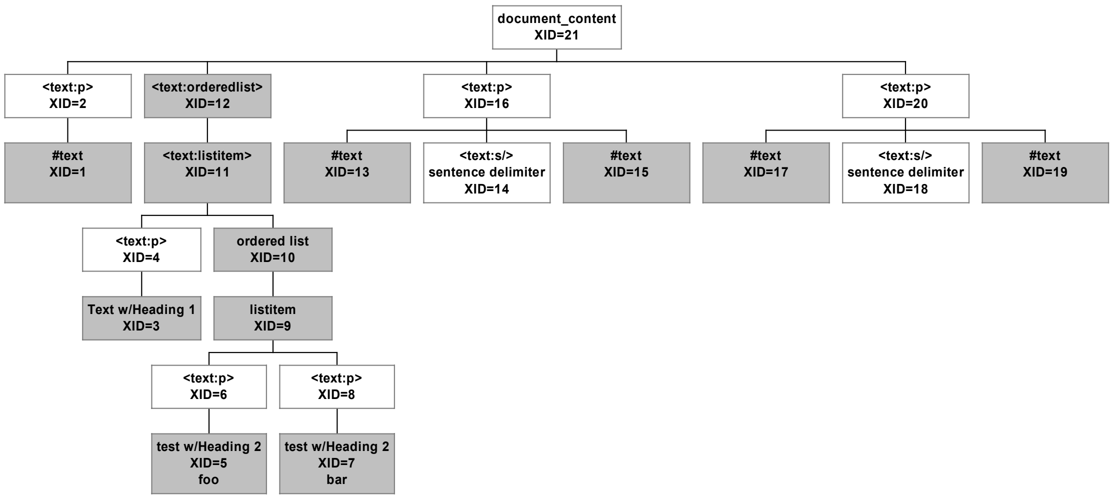

Figure 14 OpenOffice XML Structure of a Document

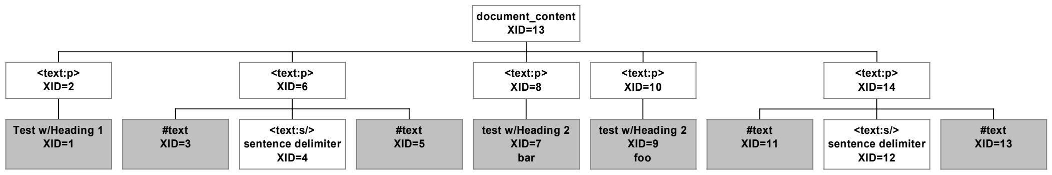

Figure 15 OpenOffice XML Imposed on a Text File

The two XML documents *vdiff()* is processing do not share the gray shaded nodes in Figure 14 and Figure 15. Suppose that the nodes labeled XID 6 in each document are matched. The simple approach of seeing if their respective parents are also matched does not work. Instead, we traverse up the v0DOM from XID 6 to the first shared node. If that shared node is not the match to the parent of XID 6 in v1, then we can call the action a strong move.

AdjustForUnSharedChildren

Like PeepHoleOptimization, this topic presents several obstacles to a generic implementation and deserves a complete review in a later section.

BuildLeastCostEditScriptForWeakMoves

Once *vdiff()* marks nodes as strong moves, insertions, and deletions we are ready for another straightforward step. The user will have left the DOM tree in a defined state upon his closing the file. Matching nodes is only one part of what *vdiff()* needs to do. The next, and potentially the most computationally expensive, step is to determine the least expensive way to get the children of a v0 node into the same order as its matched v1 node has. If we treat each child node and its sub-tree as a unit, this is simply a longest common sub-sequence problem. The task is to determine the least expensive means by which each node can turn its old child sequences into its new child sequence. The cost for inserts and deletes are proportional to the weight of the sub-tree each child represents. This process remains unchanged from *XyDiff()*.

DetectUpdatedNodes

`DetectUpdatedNodes` is also straightforward in implementation. If a node and its matched node have only singular unmatched text-node children, match the text-nodes. Consider the text-nodes updated and assign new XIDs to them. Because of our implementations difficulties with peephole optimization, *vdiff()* does not catch updated nodes unless there is only one unmatched node between the two documents.

ConductAttributeOperations

`ConductAttributeOperations` is another uncomplicated task. It is modified in *vdiff()* to take advantage of the StructuralMapInfo represented by the map file. This process looks at every matched node and determines if its attributes represent inserted, deleted, or updated values. We again use the StructuralMapInfo and its *O(1)* lookups to minimize the expense of determining if the absence of attributes in a v1DOM node are meaningful. In the domain of our test sets, this function also infers attributes for inserted nodes. This inference is critical to the proper rendering of paragraphs in both AbiWord and OpenOffice. Attributes contain the style, font, and other formatting information for every paragraph node. Overall, this process is *O(n)* with *n* the number of nodes in the v0 document.

It is now time to visit the implementation details of peephole optimization and AdjustForUnSharedChildren.

Peephole Optimization

Current Implementation

The intent of the process called peephole optimization [COB02] is to raise the number of matched nodes between the two XML documents. By increasing the number of matched nodes, we expect a corresponding decrease in the size of the edit script. The process starts with a recursive traversal the *v0* document tree and is show below.

For each matched node, collect all the unique unmatched element-node children (lines 1-10 in Figure 16). Collect all the unique unmatched element-node children of the matching node in the *v1* document. For each unmatched child with a unique element name in the *v0* child list, if the *v1* child list has at most one instance of a tag with the same label, consider those two nodes a match and give each the other's XID (lines 15-20 in Figure 16).

```
 1.   DOMNodeList getUnMatchedKids (DOMNode startNode) {
 2.   DOMNodeList childList;
 3.   childNode = startNode.getFirstChild();
 4.   while (childNode != NULL) {
 5.         if (isUniqueChildName(childNode.getNodeName() &&
 6.             childNode.matchID == NULL)
 7.             childList.addNode(childNode);
 8.             childNode = childNode.getNextSibling();
  9.        }
 10.  }

 11.  peepHoleOptimize {
 12.    if isMatched (v0Node){
 13.       v0unMatchedKidList = getUnMatchedKids (v0Node);
 14.       v1unMatchedKidsList = getUnMatchedKids (v1Node);
 15.       for (int i=0; i< v0unMatchedKidList.length(); i++){
 16.       v0kidNode = v0unMatchedKidList[i];
 17.       v1kidNode = v1unMatchedKidList.find( kidNode.getNodeName());
 18.       if (v1kidNode != NULL)
 19.          v1kidNode.matchID = v0kidNode;
 20.       }
 21.  }
```

Figure 16   Pseudo-Code for Peephole Optimization

The process is shown graphically in. The algorithm as discussed so far will match the root nodes (as a matter of necessity), and nodes 15, 17, and 26. These matches (solid lines) occur because the weights of their trees are identical or they have ID attributes—the identical XIDs in the figure are purely visual aids. Suppose the algorithm

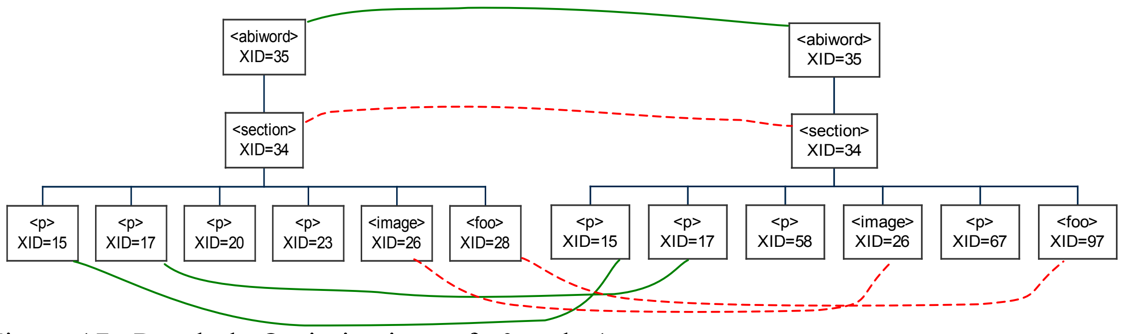

Figure 17   Peephole Optimizations of v0 and v1

does not immediately match *v0* nodes 20, 23, or 28 or *v1* nodes 58,67, and 97. Now the peephole process starts at the *v0* root and discovers it has an unmatched child, a single instance of the <section> tag. The process then moves to *v0's* root's matched node, the root of *v1*. The matched node also has only one child, also of tag <section>. The singular instance of a unique tag allows a match (the dashed line) to happen between the two <section> nodes.

Now consider the matched <section> nodes. The <section> node of *v0* has two children that are <p> nodes. Because of this, the algorithm cannot make any reasonable assumptions about which *v1* node is the respective match. In the case of the <foo> node, each <section> node has only a singular instance. Without making any inferences how far down the <foo> hierarchy this match may propagate, the algorithm makes the reasonable assumption that the two <foo> tags should be matched.

It is apparent that there remain two <p> tags in *v0* and their respective sub-trees that are eligible for matching against nodes in the *v1* DOM. As implemented in *XyDiff()*, these paragraphs will not be matched against each other, nor will their respective <c> children and text-node children. The practical effect of this is that if XID 58 should be an update action (as determined with through visual change detection) against XID 20, the algorithm will not detect it. Nor will it detect an update in XID 67 from XID 23. The other practical effect of this is that the algorithm will consider the failure to match as a delete action against XID 20 and 23, and an insertion of XID 67 and 58. This still falls into the category of sub-optimal matching when the discussion remains purely in the realm of tree matching. However, from the user's perspective in AbiWord and OpenOffice, the delete followed by an insert yields predictable consequences: the inserted paragraphs do not have the same formatting as the deleted paragraphs. From the user's viewpoint through the AbiWord GUI, it looks like all formatting beyond Style="Normal" for every changed paragraph is lost upon importing changes from the PDA to the desktop unit. This is clearly an unacceptable result and one which we discuss further.

Two Failed Approaches

We developed two further techniques in an attempt to find generic algorithms usable in the face of multiple instances of the same tag. The first was to build a list of nodes with that tag in v0 and a list for nodes with that tag in v1. The list is still exclusively of the children of the matched node (in this case, <section>). We then did a one-for-one match of each node in the lists. This method does raised the number of matches for the <p> tags, but has immediately apparent side-effects in the size of edit scripts and the resulting updated documents. These forced matches do not account for entire <p>-rooted tree's being reordered as children of <section>. Nor does the increased percentage of matches decrease the size of the edit script: it made the edit script larger in *every* case except trivial deletes with singular insertions or update actions.

We next tried to develop an algorithm that used a matched sibling as a clue towards improving peephole performance. In the same scenario as above, each unmatched child of <section> in *v0* has a choice of multiple <p> children of *v1's* <section>. We use the matched siblings of the unmatched child to help pinpoint where a likely match would be in the *v1* document. Unfortunately, this method proved incapable of coping with reordered children and inserted nodes. We did not implement this method due to these two shortcomings. A likely solution to our problem of too many choices and not enough info to make intelligent choices is in the future work chapter.

AdjustForUnSharedChildren

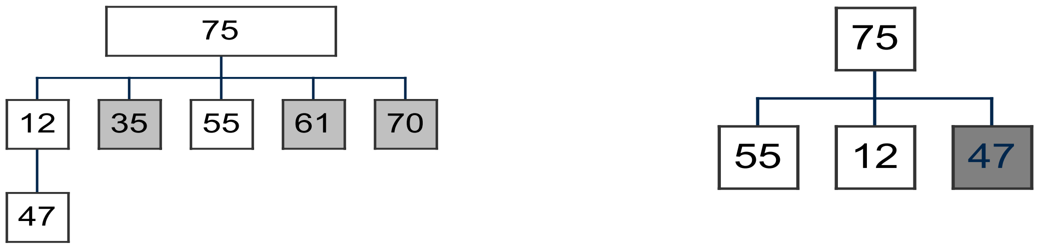

Figure 18 Simple Child Reordering Problem

An important issue is child ordering of a matched node when some of the children are not shared across XML documents. An inserted, moved, or weak-moved child may appear to be the *i*th child of a *v1* node. When incorporated back into the original document however, it should rightly be in the *j*th position. If left with an incorrect insertion point, the *vdiff()* script will insert those nodes at the point of the child list.

In the two documents shown in Figure 18, the white nodes are shared and identical numbers across both documents indicated a matched node. Light gray nodes (35, 61, and 70) are not shared across the original *v0* document and the modified *v1* document (the nodes may contain style information, pictures, or other embedded objects). The dark grey node (47) is a strong moved node: it is matched but its parent in *v0* is 12 and parent in v1 is 75. The challenge is to ensure that the ordering of paragraphs in the *v1* document stay the same with respect to each other. Another consideration is that the ordering of the unshared nodes in *v0* should also stay the same with respect to each other. Reordering the nodes without user input is unjustified and potentially ruinous of the document.

Without child reordering in *v1*, the weak move of child 55 (75's first child) would be placed first in the patched child list. The child list would look like [55, 12, 35, 61, 70]. So far this does represent a potentially fatal merge-error. When we include the insert of 47 as child 3 in the *v1* document, we end up with a child list of [55, 12, 47, 35, 61, 70]. What we are currently unable to prove is that this is “correct” in all instances. According to the original document the pattern of nodes is [S, U, S, U, U] with S representing a shared tag and U an unshared tag. When we apply the patch we have a pattern of [S, S, S, U, U, U]. Put in context we had a text paragraph, a picture, a text paragraph, and two pictures/embedded objects. After the patch we have three text paragraphs followed by three pictures/embedded objects. There are multiple ways to accomplish this merging of unshared tags and some negative repercussions in particular applications.

In the case of AbiWord (Figure 19), the <styles>, <pagesize> and other application-specific data are before all section nodes. If a user adds a new section as child 2 of *<abiword>* (), simply inserting it at position 2 in the *v0* document will break the DTD and render the document unusable.

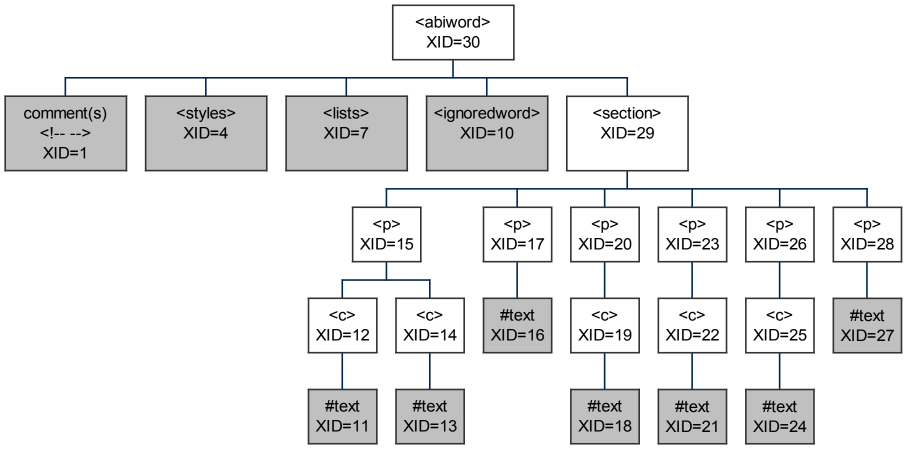

Figure 19   AbiWord XML Document With Unshared Nodes Highlighted

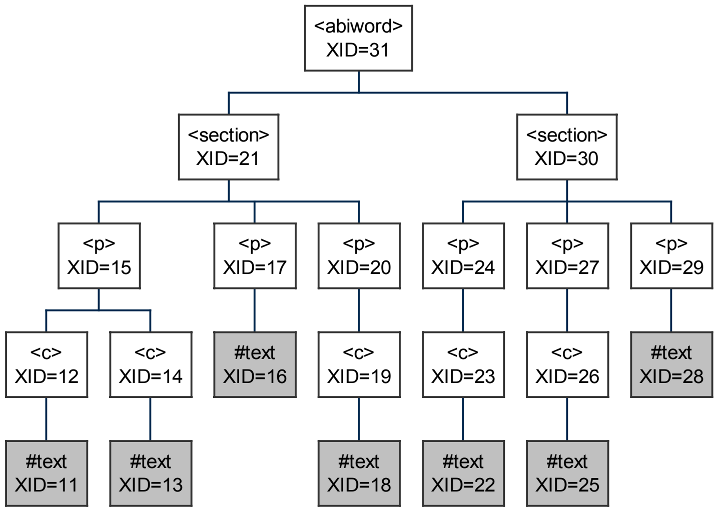

Figure 20 A *v1* Document With a Second <Section> Tag

An immediately apparent and naïve approach would be to prepend all the unshared nodes to the child list of the shared v1 parent (so 75's child list in *v1* would be [35,61,70,55,12,47]). Equally naïve would be to append the unshared nodes, giving [55,12,47,35,61,70]. In the case of AbiWord and OpenOffice documents, either approach can render the document unusable by the application.

Unsuccessful Approaches

The first approach we used was a rather convoluted dual pointer method. The matched nodes placed a pointer on each of their first children. The algorithm stepped through the v0 child list and when a node it visited was unshared, immediately copied that node into the v1 child list. The v1 child pointer moved to its next sibling only when the v0 pointer visited its matched node. When it moved, it check to see if where it was pointing had been visited yet, and if so kept moving until that condition did not hold. While this method worked in the first few test sets, it exhibited fatal behavior in a pathological case. In the event the first node in the v1 child list was the last node in the v0 child list, all unshared nodes got prepended to the v1 child list. This translated into a false move for every unshared child node, and made the document unusable or required extensive editing to fix.

Current Approach

The currently working implementation uses a simpler method, though arguably naïve in its own right. We implement a method that duplicates the pattern of Shared and Unshared nodes first displayed in the *v0* document. Let the *v0* document's matched node have a [U,U,U, S, S, U, S] pattern of child nodes. Also let the *v1* document's matched node have a pattern of [S, S, S, S]—it is impossible for the *v1* document to have unshared nodes. When reordering the children of *v1* we replicate the *v0* pattern and append any excess shared-nodes onto the end of the pattern. This method, with our test set, produced workable AbiWord and OpenOffice documents. What it failed to do was allow for not following the original pattern as defined by the v0document. We could not produce a change in the text document that increased the number of shared nodes between two unshared nodes.

Conducting Attribute Operations

The need to infer attributes for a node is entirely dependent on the class of XML documents the original document came from. Attributes in the AbiWord and OpenOffice encoding schemas allow individual paragraph level definition of their style, font, and other rendering information. In the case of AbiWord and OpenOffice, paragraphs not having attributes remain in the data file, but do not rendered correctly in the application. Other applications generating different XML documents may not need to infer attributes, and indeed may have no attributes to worry about. Before we chose to implement the intelligent default method for inserted nodes, we attempted to infer attributes of inserted nodes.

An inserted paragraph could inherit the attributes of the previous (or following) paragraph. This might be done by maintaining a list with references to every paragraph, allowing rapid movement through the sequence of paragraphs in the *v1* document without traversing the DOM tree. The time complexity of this method should be constant (though we provide no formal proof). If the program builds the list as it processes the tree, finding the previous paragraph requires a single de-reference of a pointer. Space complexity will be linear in number of unique tags in *v1* and the number of instances of each tag. Building a list, instead of maintaining information on the last paragraph, would also allow for inferring attributes across section boundaries or other portions of the document where using the last visited paragraph's attributes is insufficient.

The initial inference algorithm used a two-phase approach of copying every matched node's attributes from the *v0* document to the *v1* document. This would have the added benefit of reusing known working code from the original *XyDiff()* code that conducted attribute operations. The time complexity of this step was O(*n)* with *n* the number of matched nodes. The second phase used a pre-order traversal of the *v1* DOM-tree to find the previous paragraph and copy the attributes to the *v1* node. This two-phase approach induced experimental processing times an order of magnitude higher than using the default attribute method. It also caused attribute operations to start dominating the total processing time as the number of inserts in a document climbed. We will show in the results chapter why we abandoned this technique. Post-experimental analysis also revealed that the implementation was actually O(*nm*) with *n* being the number of nodes inserted and *m* being the number of nodes in the *v1* document. Traversing *m* nodes for each of *n* inserted nodes caused a tremendous negative impact on the processing time.

Another drawback of inheriting from previous paragraphs is it makes no sense when the new paragraph belongs to a different chapter, section, or other boundary within a document. A final drawback of inheriting attributes is the domain specificity of the technique: its okay for paragraphs in word processors, not for a generic XML document with no knowledge of the knowledge domain the document belongs to.

We developed another a single-phase approach to attribute inference that also relied on inheriting from previous or following paragraphs. Instead of building a data-structure of unique tag names and lists of nodes with that tag, we used the DOM Node's ability to traverse from sibling to sibling. This proved insufficient to always find a previous paragraph node at the same level in the DOM tree: consequently demanding a reverse-order traversal starting at the node we needed to get attributes for. We believe this reverse order traversal, also worst-case O(n), would lead us to another O(nm), for m inserted nodes. We did not implement this method as one O(nm) solution already proved disastrous in performance.

CHAPTER 6
EXPERIMENTAL RESULTS

Test Platform, Equipment, and Methodology

The testing platform was a PC equipped with 128 MB RAM, an 800 MHz Pentium III processor, and a Maxtor 52049H3 Hard Disk. It was running RedHat Linux with kernel 2.2.16, and Xerces-C++ 1.4. We used gmake 3.79.1 and gcc/g++ 2.96 to compile the program and its associated utilities. We also used a Sony Vaio Pentium III with 256 MB RAM also running Red Hat 7.2. The Sony used the same version of gmake and gcc/g++ 3.04 as its compiler.

Test data for the experiments consisted of a just over a dozen term papers, memos, letters, and scratch documents. We used AbiWord [SOU02] and OpenOffice's StarWriter [OPE02] word processors to create the documents and save them to their canonical XML format. Since both applications store document data in XML this made no conversion from proprietary formats necessary. The test documents at this stage contained no graphics or other embedded objects, just formatted and styled text. We deliberately kept the test set small to better control the fluctuation of document structure within a collection of word processing documents.

We then used an automated script to insert, delete, move, and update 10%, 20%, 30%, 40%, 50%, 60%, 70% and 80% of the paragraphs of each originating document. The sequence of testing shown below graphically shows what the following verbiage describes.

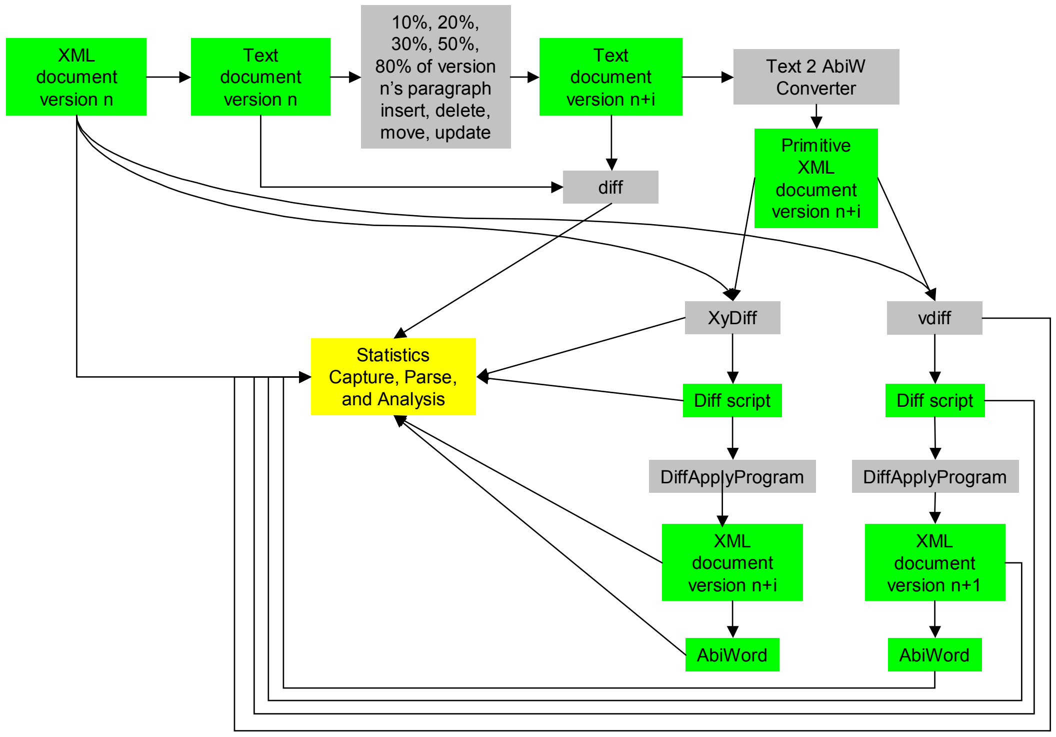

Figure 21 Testing Methodology for *vdiff()*, *XyDiff()*, and *diff()*.

The quantity of data needed from multiple sources mandates an automated test setup. The overall script controls the generation of test documents from the original XML. It converts the XML into text, then executes the modifications to 32 copies of the text file. The script then runs *diff()* against the text only files. It also runs both *XyDiff()* and *vdiff()* against the original XML and the primitive XML files. Next it applies the edit script to update the original XML file. The script captures all the statistics associated with each of these tasks. In addition, we must visually inspect the AbiWord document the patch program created. This allows a user's-perspective of changed, lost, or misplaced paragraphs and their formatting through the AbiWord interface.

Abiword as Data Generator

A side effect of using the AbiWord Word processor as a generator of test documents presented itself very early. The DTD that all AbiWord files reference is not up-to-date with respect to the files that AbiWord generates. AbiWord uses a custom built parser that does not validate the input documents against the DTD. Our solution, after consultations with the AbiWord Development team, was to remove the DTD reference from the data file. The practical result is that removal of the DTD reference does not interfere with reloading the document into XML.

The prime difficulty with AbiWord in this research is the sporadic inconsistency in embedding a paragraph tag within appropriate tags. As discussed earlier, most paragraphs have a structure that looks like <p><c>text stuff</c></p>. Approximately five percent of the time, AbiWord instead generates paragraph structures like <p> text stuff </p> or even <p> text stuff <c></c></p>. Discussion with the AbiWord development team is ongoing and has not resolved the unpredictable nature of this problem. When this phenomena appears, we have either left it as is, and labeled the visual error it produces as an error, or we have normalized the structure to the <p><c>text</c></p> structure.

OpenOffice Writer as Data Generator

OpenOffice is a direct descendant to Sun's StarOffice 5.2. As such, OpenOffice as a significant more polished and refined user interface. OpenOffice also has a richer complement of currently implemented features than AbiWord. This feature set provides the ability to expand the types of structures embedded in the documents and stretch our format conversion techniques.

Another advantage of OpenOffice is it is an office suite of tools, not a single application that generates word processing documents only. OpenOffice has a presentation application much like Microsoft's PowerPoint. OpenOffice also has a spreadsheet application. Yet another important advantage of OpenOffice is that all its applications use XML encoding for all their data formats. The XML reside in ZIP archives that contain additional data about each file, but accessing the XML content is supremely convenient.

Results

Bandwidth Conservation Through Content Conversion

One of the propositions this research puts forward is that there is a quantitative difference in the amount of data a device must transfer for a MS-Word document compared to an ASCII document. Though this seems like common sense, quantifying the actual values gives a better knowledge of the actual savings. Research with the Puppeteer [FLI01] system also validates the savings information listed below.

The first place to see the quantitative difference between documents is to look at the rather widespread Microsoft Word application. To gather this data, we searched the personal home computer of the author and discovered 398 MS Word documents. The documents run a gamut from single page letters to businesses and friends to organizational manuals, term papers, and thesis documents.

Table 3 Average Microsoft and Text Document Size

	MS-Word	ASCII Text	Ratio
Average Size	38.8 KB	7.4 KB	0.19
Std Deviation	22.5 KB	10.3 KB	
Average Savings		31.4 KB	

To continue this examination of savings, we examined the test set of AbiWord documents. This set contains 13 documents created by importing 13 different word documents from the set used in Table 1.

Table 4 Average AbiWord and Text Document Size

	AbiWord	ASCII Text	Ratio
Average Size	23.6 KB	6.3 KB	0.27
Std Deviation	16.3 KB	4.8 KB	
Average Savings		17.3 KB	

This is a significantly smaller test set than that created for Table 1 but the savings are still plain. Graphically we depict this information below in Figure 2 through Figure 5.

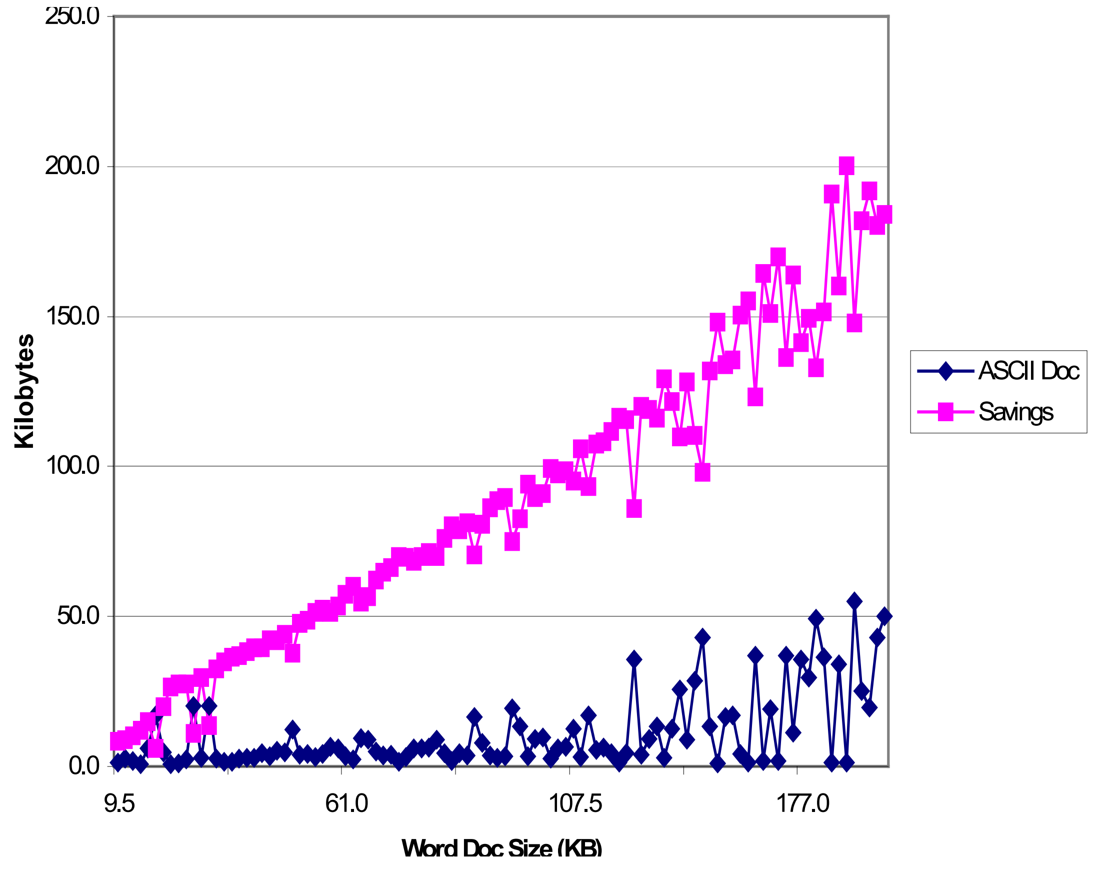

Figure 22 Comparative File Size for MS-Word Documents and Text Documents

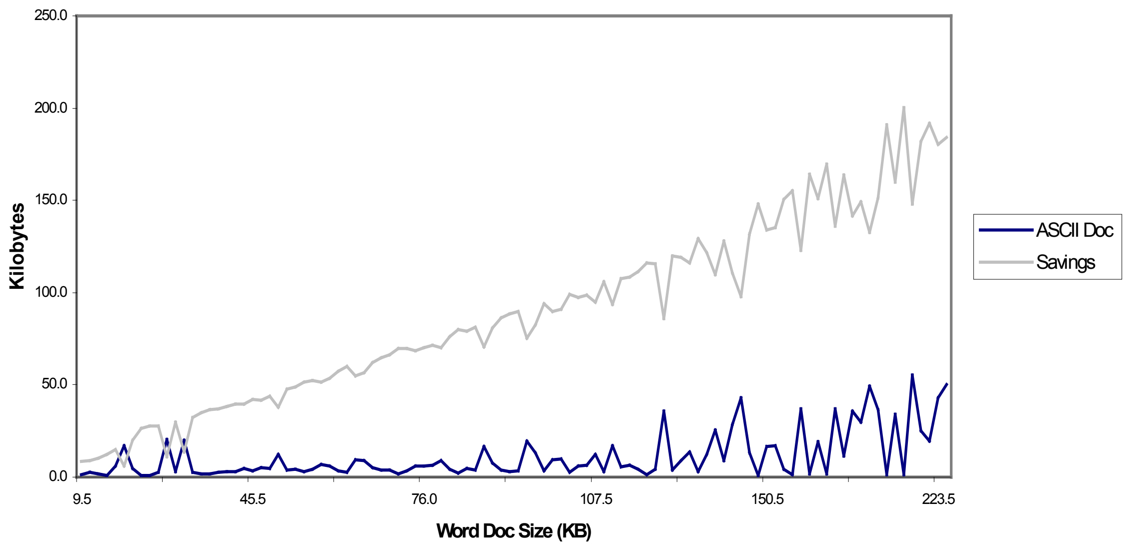

Figure 23 Text Document and Byte Savings as Word Documents Increase in Size

Finally we show the standard deviation in the size of the ASCII documents. This provides us with more justification to believe that the size of the text documents stays relatively constant. The larger a Word document the more often it has multiple embedded objects in it. It is those embedded objects that produce the majority of the growth in Word documents: it is not the text component of documents. This information also indicates that in-memory processing of the documents will not present a significant burden to PDAs. Even small PDAs now how upwards of 8 MB of memory built in.

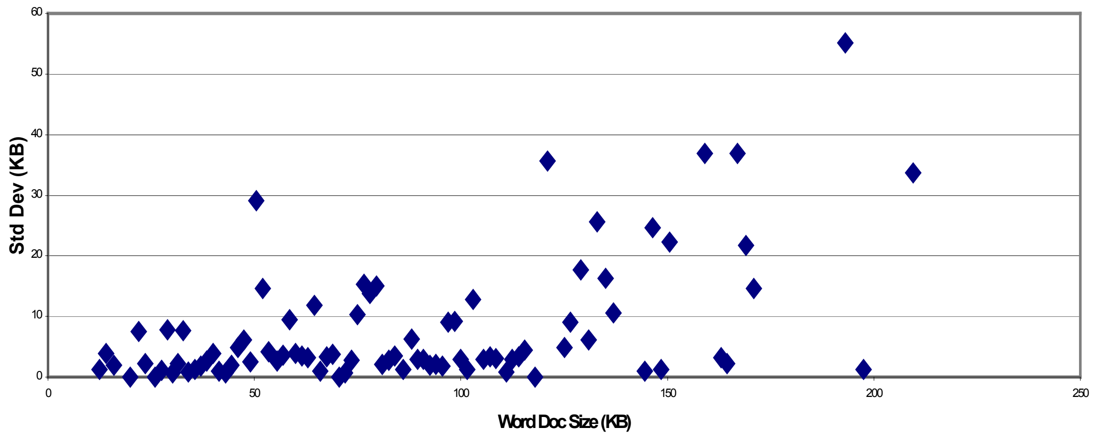

Figure 24 Standard Deviation of ASCII Documents Derived From Word Documents

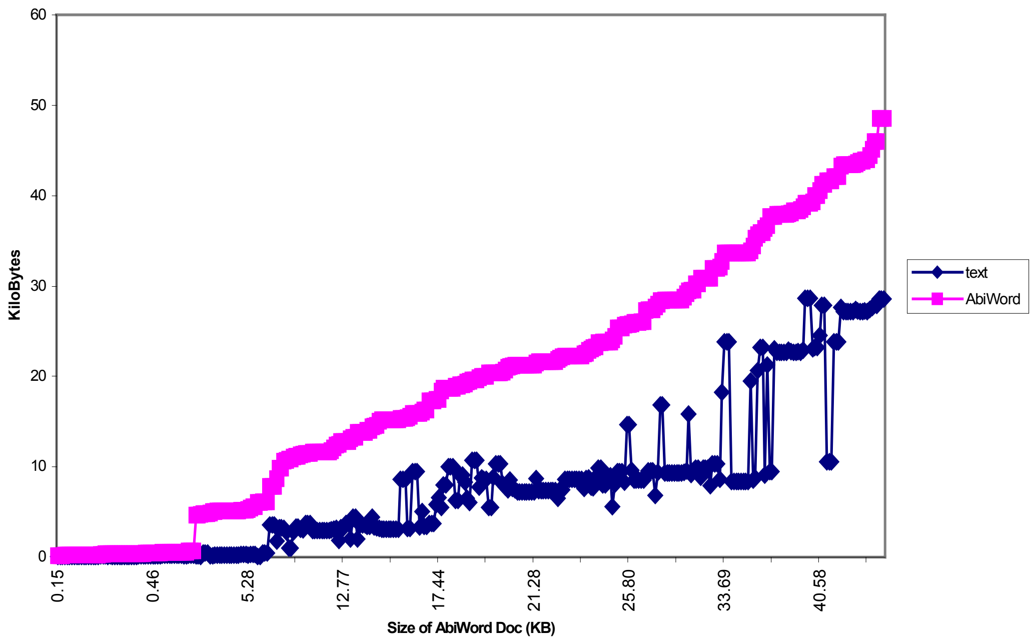

Figure 25 File Size for AbiWord Documents and ASCII Text Documents

Bandwidth Conservation Through Client-Side Use of GNU *diff()*

When the mobile devices update data they have two choices on how to transmit those changes back to the UbiData server. The first choice is value shipping and the second choice is shipping an edit script. Using information from above we can see that sending entire documents while fail to meet any of the UbiData goals.

For mobile devices incapable of providing the processing power and hardware to support *vdiff()*, we have designed the system so that the receive text only documents. When the mobile devices needs to ship the changes back to UbiData, it uses the text only *diff()*. The table below shows the summarized version of the savings we incur by using *diff()* compared to value shipping. It also shows summary data of information presented in the next section: *vdiff()* versus *XyDiff()* versus *diff()*.

Table 5 Savings Achieved Through Use of *diff()* Tools versus Value Shipping

	Value Shipping	*diff()* Generated Script	*vdiff()* Generated Script	*XyDiff()* Generated Script
Avg file size	5735 Bytes	1063 Bytes	2039 Bytes	7679 Bytes

The same information displayed graphically is below in Figure 6. The mobile client will be able to send significantly smaller quantities of data by taking advantage of *diff()*. Without using a change detection mechanism, the client would have to ship the entire file anytime it needed to update the server.

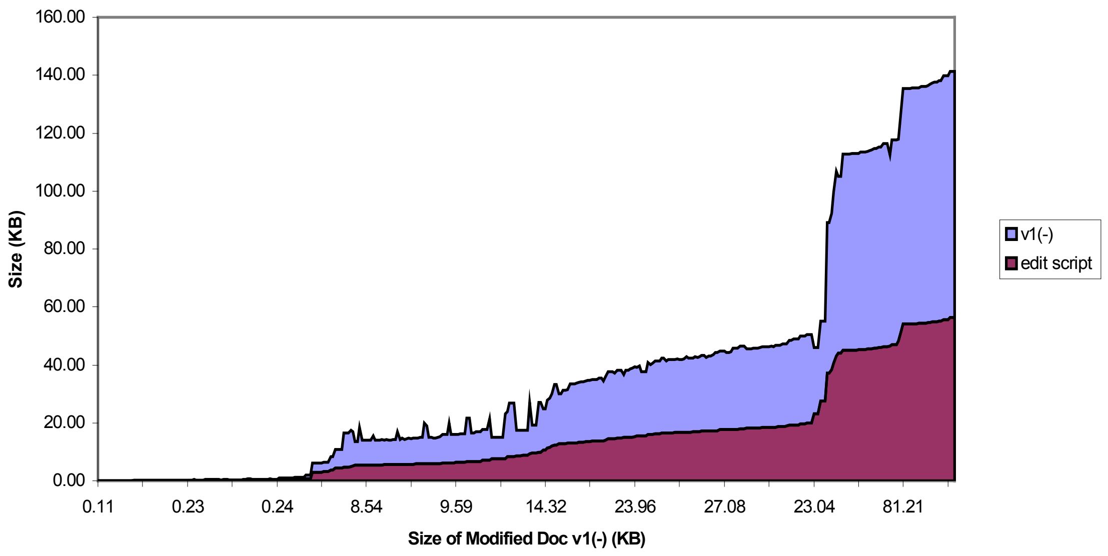

Figure 26 Savings Through *diff()* on Client versus Value-Shipping Entire File

The size of our test set is too small to confirm these ratios will hold across the entire spectrum of documents that AbiWord and OpenOffice can create.

Comparison of *vdiff(), XyDiff(), and* GNU *diff()*

Our vdiff() performed consistently faster than *XyDiff()* despite the extra overhead associated with the `StructuralMapInfo` discussed in the Implementation Chapter. The likely cause of this behavior is the increased costs *XyDiff()* incurs by treating every absence of a node in *v1* that was originally in *v0* as a delete. As we discussed, some nodes are not in *v1* because of deliberate decisions to curtail the amount of material transmitted to the mobile client. *XyDiff()* must create additional entries in the edit script and must write to disk a larger edit script than *vdiff()*.

Prior to going further we need to explain the use of *diff()* as a benchmark by which to put our performance in context. In some ways, comparing *diff()*'s performance against *vdiff()* and *XyDiff()* is inappropriate. It is similar to comparing the efficiency by which a hammer inserts screws into a piece of wood.

The first purpose provides a differential look at the amount of time a mobile device will commit to processing a text-only *diff()* on text-only files compared to an XML *diff()* on XML files. Increased processing demands will necessarily decrease the battery life of a mobile device: use of text-only files with text-only *diff()* may help limit the amount of processing power a device devotes to this activity. This viewpoint is shown in Table 3 previous section and substantiated by the tiny *diff()* script size in the graph above.

The second purpose of using *diff()* as a benchmark is to show that *vdiff()*, though not as fast as *diff()*, is still faster than other current technology. We have made progress by improving not only performance in time but also in the size of *diff()* scripts generated.

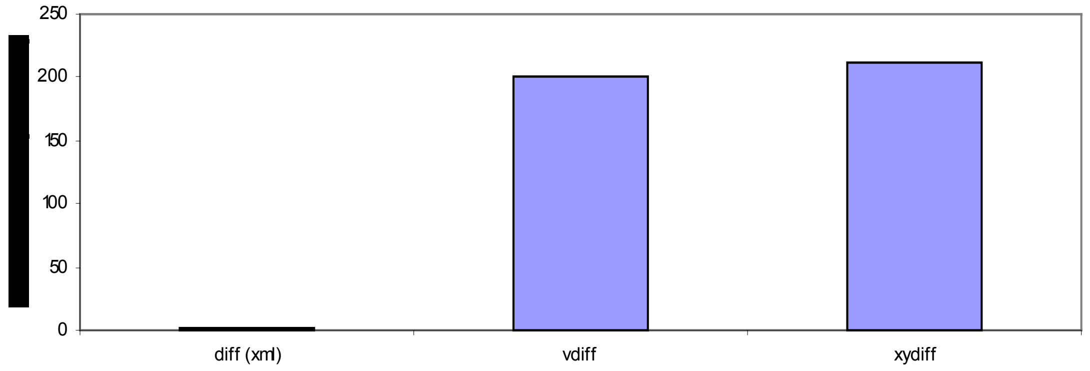

Figure 27 Cumulative Execution Times for *diff()*, *vdiff()*, and *XyDiff()*

The time required by *diff()* to execute its algorithm on the text only documents is near constant within the test set. The time required by *diff()* to execute on pre-processed *v0* and v1(-)' is also near constant as shown below in Figure 8. It shows that the GNU *diff()* clearly outperforms *vdiff()* and *XyDiff().* It also shows that *vdiff()* does outperform *XyDiff()* as shown in Figure 7.

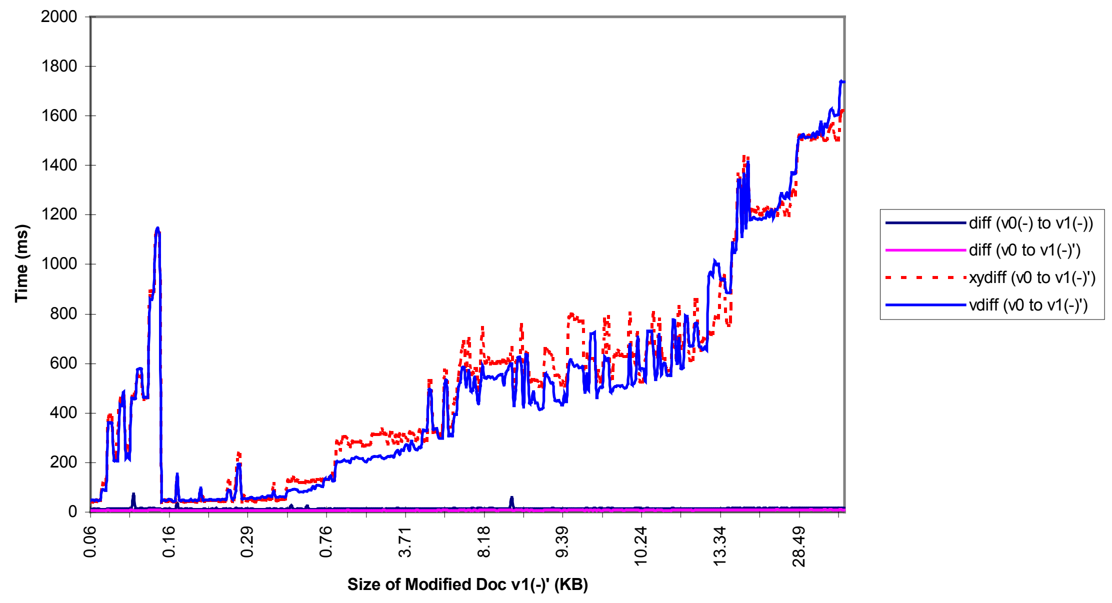

Figure 28 Comparative Execution Time for *diff(), vdiff(), XyDiff()*

The information in Figure 9 also shows a clear improvement over existing technology. Though we again do not meet the standard that *diff()* sets, neither do hammers often succeed in correctly setting screws into woodwork. We do however offer a power drill with screw driver attachment while *XyDiff()* offers the manual screwdriver.

There are two critical distinctions to keep in mind when reviewing the large *diff()* script size relative to *diff()* and the slow execution time relative to *diff()*. The first consideration occurs when GNU *diff()* runs against the full-content XML document and the primitive XML document sent back from the mobile device. The edit script it will cause removal of all application specific data that provide key and essential rendering information to AbiWord and OpenOffice. It will corrupt the document such that the file becomes unusable. The second consideration comes into play when using *diff()* against actual XML files, and not pure text files as shown above. XML is text and a natural assumption would be to use *diff()* on this text file. The difficulty arises when the XML file is only one or two lines long, but has kilobytes or megabytes of data within those two lines. A minor change in a single node will require *diff()* to generate a script twice the size of the file.

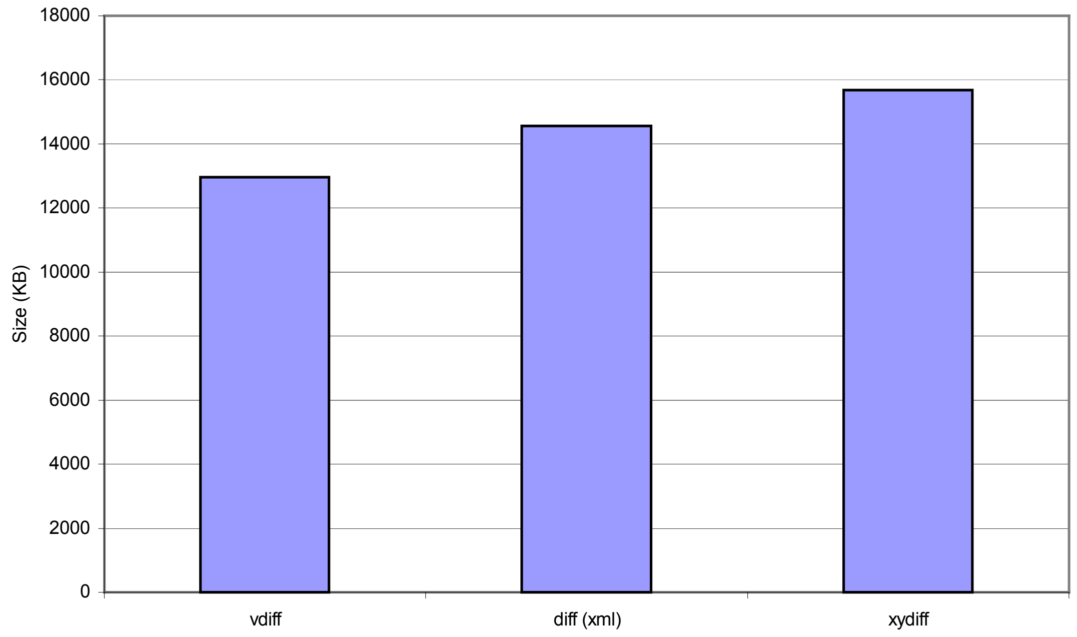

Figure 29 Delta Script Sizes for *diff()*, *vdiff()*, and *XyDiff()*

One other measure of performance we want to convey is the number and types of errors *XyDiff()* and *vdiff()* induce in AbiWord documents. It is possible, even likely, that the two tools will generate errors in the matches. Most current research typically refers to these types of errors as sub-optimal matching. It is also possible that the results of the sub-optimal matching will have no impact at the application and user level. A prime example is the false movement of empty lines. These false moves are erroneous: they cause larger *diff()* scripts and non-optimal matching. From the user's viewpoint, working through the AbiWord interface, no error is apparent. Thus errors in *vdiff()* matching must receive two standards by which to judge their quantity and severity. Those two standards are in terms of non-optimal matches and in terms of user identifiable errors.

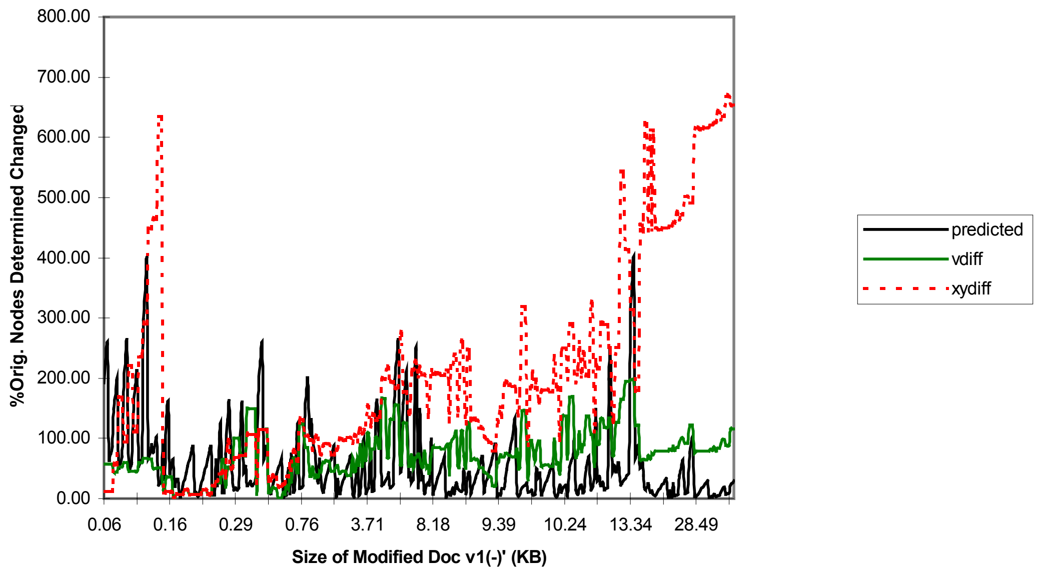

Figure 30 Predicted Performance Versus Actual *vdiff()* and *XyDiff()* Performance

The three lines of performance in Figure 10 reflect the first standard of measurement: non-optimal matching. Under an ideal and optimal matching, the expectation is the number of nodes *XyDiff()* and *vdiff()* identify as changed should match the number we know are changed. We know precisely the number of nodes that get changed by insertion, deletion, movement and updating paragraphs. With this knowledge the predicted line comes into existence. We gather the data from *XyDiff()* and *vdiff()* themselves on the number of nodes they identify as changed with respect to the version *n* XML document.

With the three sets of data, we can show that *vdiff()* outperforms *XyDiff()*. We also see that *vdiff()* is less than optimal. There are two principle causes for the less than optimal performance. The first is that the algorithm does not consistently identify updated nodes. Unless there is only one paragraph updated, the algorithm is able to find it. If there is more than one, *vdiff()* cannot currently determine which *v0* paragraphs match which *v1* paragraphs. This implementation problem is thoroughly discussed in the implementation section of this thesis and in the future work section. The second performance problem is caused by false moves of empty lines represented by empty paragraph tags (<p/>). The lookup table created when traversing the v0 document inserts XIDs into a vector identified by the hash value of the node. All empty paragraphs throughout the entire document have identical hash values. Since the XIDs are stored in a vector, we end up with a first visited-first matched situation for all empty lines in *v1* against all empty lines in *v0*. The last measure we will discuss is the number and type of errors a user perceives in AbiWord.

Recall that a goal of UbiData is to allow the executive to make spelling error corrections and other edits on his mobile device. With this system he is supposed to have a high level of confidence that his changes will correctly propagate to the server. This expectation leads us to our last measure of performance for *vdiff()*. The information shown in represents the number of errors *vdiff()* induces in documents that have edit scripts applied to them. All the errors associated with *vdiff()* originate with the update detection mechanism. As update does not work, *vdiff()* treats all updates as deletes followed by inserts. Consequently, all updated paragraphs get reinserted using the default attributes of "Normal." The information in Figure 11 can be slightly misleading. Recall that AbiWord is unable to open or otherwise use the *v1* files created by XyDiff(). The line in the figure below represents the number of paragraphs in the test documents.

Each of those paragraphs would have lost all formatting that may have existed in *v0*: hence the extrapolated data points for errors generated by AbiWord.

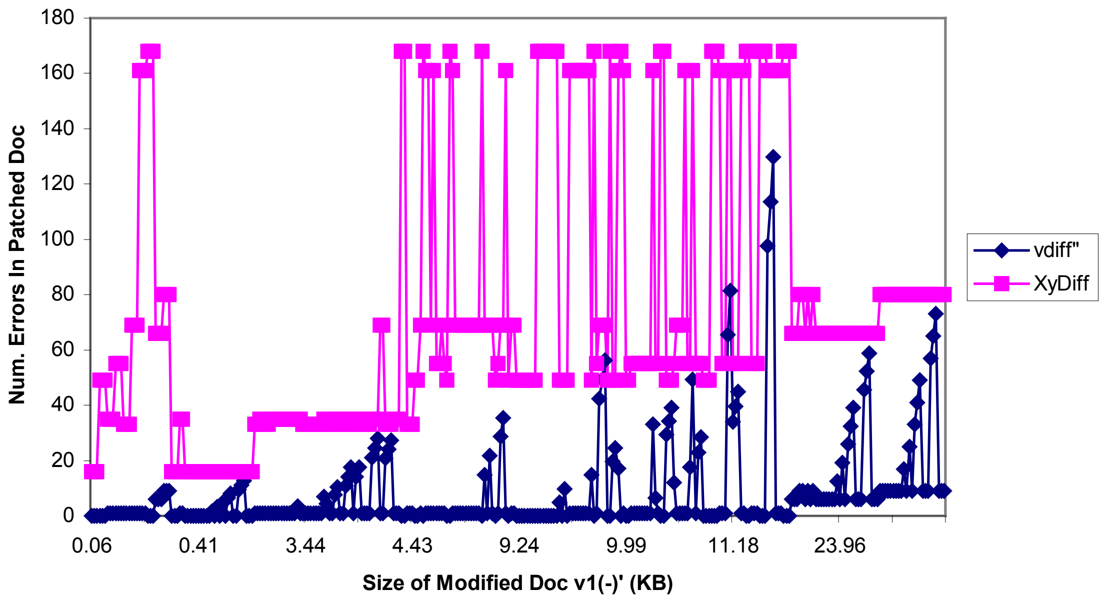

Figure 31 User Perceived Errors in *v1* Documents Using AbiWord

CHAPTER 7
FUTURE RESEARCH

Peephole Optimization

The most likely solution to improving the number of matches *vdiff()* discovers between two documents is to add another step of processing before trying the peephole method. After more experiments and brainstorming, we have come to the conclusion that one-layer deep optimization is the best an algorithm can do when it knows nothing of the defined structure.

It would be possible, and preferable for a peephole technique on Abiword documents be different than the technique for OpenOffice. While contrary to programming to the generic solution, this custom approach should overcome the hurdles and performance penalties of the *XyDiff()* method.

Specifically there should be two lines of effort. One will involve expanding the schema that tracks structural similarity to include information a generic peephole method can use. In the case of AbiWord, we can encode information that <p> tags and their immediate <c> children cannot provide sufficient information to help. Indeed, the peephole algorithm must go to the <p> and <c> tags' text nodes and use their content as a basis to judge similarity to other <p> tags' text descendants.

Once comparing the text nodes, we can adapt current work on approximate string matching or even use GNU *diff()* to determine the likely candidates for a match. If two text nodes cross a threshold of similarity, again defined in the schema, then *vdiff()* can declare them a match and mark the nodes for an update. The effect will be an increase in the number of matched nodes between the two documents. With the increase in matched nodes and nodes declared updates, we will have a corresponding decrease in the umber of delete/insert pairs.

An additional effect of this modification will be to reduce the size of the old and new data embedded in the Update actions of the *diff()* script. Update actions in the *diff()* script carries with them the old and new values of a node. Since we have used a *diff()* tool to measure similarity we can take the output of that tool and use it as the data stored in the *diff()* script. This stands in contrast to storing the entire old and new paragraph: which is the current method.

An alternative to doing top-down optimization is to start at the leaf nodes and again use some threshold of similarity defined in the schema. When two nodes cross the threshold, consider them matched and attempt to propagate their match upwards among their respective ancestors. This technique would be very similar to that used in [CHA96] and in the *vdiff()* and *XyDiff()* `MatchUsingIDAttrs` method.

Child Reordering

After conducting over 100 separate tests, the evidence does not support the ability to do this in a provably correct manner. There will always be cases where the original sequence of unshared and shared nodes ultimately has little bearing on how a user wants a document to look.

To successfully and provable maintain correct child ordering, all the children need to be at the disposal of the user (directly or indirectly). We have shown the savings capable by not transmitting all the content of an XML document to a limited capability device. Instead of loosing that advantage, use placeholders to mark the location and other vital information of large embedded objects. Like Puppeteer we will transmit low-fidelity images to mobile devices that can use the degraded content. Examples would be HTML style links to embedded objects, bounding boxes for graphics, and stylized text markup for ASCII editors.

The existence of the placeholders will almost obviate the need for the structural similarity map. However, there will be some content that will still not make sense to transmit and certainly not make sense to expose the user to. Good examples are the style nodes at the front of AbiWord and OpenOffice. Since they all exist, indeed, must exist, at the front of their respective documents, reordering of unshared and shared children becomes trivial.

Optimizations of Current Implementation

Edit **script overhead**

The edit scripts generated by the current implementation use some what verbose tags. The tags make it very easy for a human to read the script, but impose a certain amount of overhead. Experiments from the first set of experiments showed the potential for a 5% reduction in edit script size by redesigning the script schema.

Edit s**cript packaging of old data**

One of the principle advantages of properly constructed edit scripts is the ability to time-shift the current version of a document. A user can subtract edit script from a current version and recreated a version of a document that existed in the past. Every difference detection tool researched stores all the changed data in the edit script. A seemingly reasonable approach may involve taking advantage of the server within the UbiData environment.

The argument exists that the old values of a deleted or changed node do not need to reside in the edit script. We can offload that storage requirement to the server’s disks when it receives an edit from a client. Mobile clients with the processing power and hardware needed to execute *vdiff()* will not be using text-only versions of documents. They may use content reduced to save time downloading.

A client reports via the *vdiff()* generated edit script that it has deleted and/or modified certain nodes. In the case of delete, the client must only communicate the delete of XID n. On receipt of that action, the server stores the old contents of XID n in a file maintaining all the XID, value pairs of deleted and updated nodes. For an update, the server likewise saves the old value and XID before applying the new value. This shifting of data storage from the *diff()* script to the server that patches the master files will not save disk space. It will however reduce the amount of data that the *diff()* script must contain.

XML to HTML and back

Our goal has been to allow users to edit content on devices of their choice in applications of their choice. One possible way of allowing cross application sharing and editing of documents lies in the use of HTML as the common format between all the word processing applications. The user may or may not be aware that the underlying format of his document is HTML instead of MS Word or WordPerfect. What they see in their editor is the rendered version of the original XML document.

Immediate difficulties can be seen with using HTML as the common format. In particular is the likely requirement that multiple XML tags get mapped to the same HTML tag. While the Structural Similarity map can help determine when not to delete a node from the v0 DOM, it may be insufficient to convert from HTML back to XML. It is possible that we can embed enough hidden attributes in the HTML tags to allow for an easy reverse transformation to XML and the XML *diff()* that needs to occur. Failing the ability to embed hints in the HTML file the bottom up optimization discussed earlier may help. When text nodes get matched, it provides very good clues to which parents are good candidates for a match.

There are also known limitations of HTML to render a word processing document correctly. Their point of origin is different (one is page oriented and the other is screen oriented) and one has a richer feature set than the other. Users who wish to enjoy the liberties of sharing data across unlike machines may need to adjust their habits to accommodate their equipment's performance.

.

CHAPTER 8
CONCLUSIONS

This research has presented change detection and propagation methods to synchronize documents stored on various computing devices. This approach is capable of computing changes a user makes in one document and applying those changes on another document. The uniqueness of this capability is the changes are on a document in one format, and the edit script is applied to a document in another format. This cross-format change detection and propagation allows editing documents on simple devices even though the document originates from powerful applications on desktop and server environments.

This work contributes to the state-of-the-art in the following important ways. It presents algorithms and techniques for reducing and transforming rich content XML documents into mobile device usable forms (specifically into ASCII text for use by a text editor). The work also presents methods to convert the changed mobile device document into a form usable by a sophisticated change detection tool—we can impose a primitive XML structure on the text document. We developed tools to track how the primitive XML and rich-content XML tag and attribute sets relate to each other and inform the change detection engine what changes can be meaningful. The techniques also includes heuristics on inferring (when appropriate) default attributes for inserted nodes and for ensuring meaningful ordering of the changed document's content.

This work comprises several first steps toward realizing the goals of UbiData: anytime, anywhere access to data; device independent access to data; and application-independent access to data.

Content Conversion

One of our operating assumptions is that conversion of proprietary data formats into XML is a feasible task. Since we use AbiWord and OpenOffice to generate out XML we were able to bypass this question. Microsoft has released OLE APIs that allow conversion of Microsoft Office 2000 and Microsoft Office XP documents and structures into their XML representations [DEL00]. As Microsoft has a dominant position in the office suite of applications, adapting those programs to store data in a canonical form will greatly broaden the potential reach of UbiData.

The use of an XML canonical format is essential to UbiData's goal of cross-application portability of changes to documents and data. Continued effort must develop more sophisticated tools to convert proprietary and widespread formats into the UbiData model.

Content Reduction

Content reduction in UbiData is a static process. Unless a radical addition to the architecture occurs, UbiData will maintain only this static capability when hoarding files to mobile devices. To meet the overall intent of UbiData as envisioned in the National Science Foundation proposal, adaptation of Puppeteer techniques should be a design goal.

XML Differencing

Completely generic XML algorithms will probably not work efficiently in the environment we envision for UbiData. The different document structures with OpenOffice and AbiWord provide proof that efficiencies can improve by improving the granularity of matching. In particular, AbiWord encodes entire paragraphs under one or a few tags. Instead of limiting ourselves to the AbiWord document structure, we know that a better match (better in terms of size of *diff()* produced) can occur by performing approximate string matching against the unmatched paragraphs.

Another fundamental drawback to generic XML *diff()* solutions is their deletion of unconverted data. Modifications to prevent this are fairly straight forward. As the complexities of the originating documents rise, it is essential to solve the child reordering problem with or without the use of placeholders. It is also possible that a hybrid approach will solve child reordering by putting placeholders only between shared tags: all unshared tags that happen before any unshared nodes can then be prepended to the child list.

Continued efforts at meeting the challenges of UbiData are well worth the effort. The world continues to become more mobile in terms of people, work, computing, and communications. Failure to achieve more seamless and automated integration of our various working sets will cause ever-growing losses in productivity. As the economy grows ever more competitive, lost productivity leads to lost economic opportunities and growth for people and organizations.

LIST OF REFERENCES

[AHM95] Ahmad, T., M. Clary, O. Densmore, S. Gadol, A. Keller and R. Pang. "The DIANA Approach to Mobile Computing." Presented at *MOBIDATA: NSF Workshop on Mobile and Wireless Information Systems*. Rutgers University, Brunswick, NJ. May 1995. 15 May 2002 <http://www-db.stanford.edu/pub/keller/1994/diana-mobidata-short.pdf>.

[APA02a] Apache Software Foundation. "Xalan C++ XSLT Processor." 15 May 2002 <http://xml.apache.org.xalan-c/index.html>.

[APA02b] Apache Software Foundation. "Xerces C++ Parser." 15 May 2002 <http://xml.apache.org/xerces-c/index.html>.

[BRO95] Brockschmidt, K. *Inside OLE*. Redmond, Washington: Microsoft Press, 1995.

[CHA95] Chappell, D. *Understanding ActiveX and OLE.* Redmond, Washington: Microsoft Press, 1995.

[CHA96] Chawathe, S. S., A. Rajaraman, H. Garcia-Molina, and J. Widom. "Change detection in hierarchically structured information." Presented at *ACM SIGMOD International Conference on Management of Data.* Montreal, Quebec, Canada. 4-6 June 1996. 15 May 2002 <http://www-db.stanford.edu/c3/papers/html/tdiff3-8/tdiff3-8.html>.

[COB02] Cobéna, G., S. Abiteboul, A. Marian. "Detecting Changes in XML Documents." Presented at the *International Conference on Data Engineering.* San Jose, California. 26 February-1 March 2002. 15 May 2002 <http://www-rocq.inria.fr/~cobena/cdrom/www/xydiff/eng.htm>.

[DEC95] Decouchant, D., V. Quint, M. Romero Salcedo. "Structured Cooperative Authoring on the World Wide Web." Presented at the *Fourth International World Wide Web Conference.* Boston Massachusetts. 11-14 December 1995. 15 May 2002 <http://www.w3.org/Conferences/WWW4/Papers/91>.

[DEL00] de Lara, E., D. Wallach, W. Zwaenepoel. "Opportunities for Bandwidth Adaptation in Microsoft Office Documents." Presented at the *4th USENIX Windows Systems Symposium*. Seattle, Washington. 3-4 August 2000. 15 May 2002 <http://www.cs.rice.edu/~delara/papers/usenix_win2000/index.html>.

[DEL01a] de Lara, E., R. Kumar, D. Wallach, and W. Zwaenepoel. "Position Summary: Architectures for Adaptation Systems." Presented at *Eighth IEEE Workshop on Hot Topics in Operating Systems (HotOS-VIII)*, Schloss Elmau, Germany. May 2001. 15 May 2002 <http://www.cs.rice.edu/~delara/papers/hotos2001/index. html>.

[DEL01b] de Lara, E., D. Wallach, W. Zwaenepoel. "Collaboration and Document Editing on Bandwidth-Limited Devices." Presented at *Workshop on Application Models and Programming Tools for Ubiquitous Computing (UbiTools'01).* Atlanta, Georgia. September 2001. 15 May 2002 <http://www.cs.rice.edu/ ~delara/papers/ubitools/index.html>.

[FLI01] Flinn, J., E. de Lara, M. Satyanarayanan, D. Wallach, and W. Zwaenepoel. "Reducing the Energy Usage of Office Applications." Presented at *IFIP/ACM International Conference on Distributed Systems Platforms (Middleware 2001).* Heidelberg, Germany. 12-16 November 2001. 15 May 2002 <http://www.cs. rice.edu/~delara/papers/middleware2001/index.html>.

[FSF02a] Free Software Foundation. "Flex." 15 May 2002 <http://www.gnu.org/ software/flex/flex.html>.

[FSF02b] Free Software Foundation. "Diffutils." 15 May 2002 <http://www.gnu.org/ software/diffutils/diffutils.html>.

[HEL01] A. Helal, J. Hammer, A. Khushraj, and J. Zhang. "A Three-tier Architecture for Ubiquitous Data Access." Presented at *First ACS/IEEE International Conference on Computer Systems and Applications*. Beirut, Lebanon. 2001. 15 May 2002 <http://citeseer.nj.nec.com/rd/80023731%2C498381%2C1%2C0.25% 2CDownload/http%253A%252F%252Fciteseer.nj.nec.com/cache/papers/cs/2499 1/http%253AzSzzSzwww.harris.cise.ufl.eduzSzprojectszSzpublicationszSz3tier.p df/a-three-tier-architecture.pdf>.

[IBM02a] IBM Corp. "XML Diff and Merge Tool." 15 May 2002 <http://www. alphaworks.ibm.com/tech/xmldiffmerge>.

[IBM02b] IBM Corp. "Pervasive Computing." 15 May 2002 <http://www-3.ibm.com/ pvc/pervasive.shtml >.

[JON94] Jones International and Jones Digital Century. "Osborne Computer Corporation." *Jones Telecommunications and Multimedia Encyclopedia*. 1994. 15 May 2002 <http://www.digitalcentury.com/encyclo/update/osborne.html>.

[LI02] Li, D. "Sharing Single User Editors by Intelligent Collaboration Transparency." Presented at the *Third Annual Collaborative Editing Workshop, ACM Group.* Boulder, Colorado. 30 September 2002. 15 May 2002 <http://www.research.umbc.edu/~jcampbel/Group01/Li_iwces3.pdf>.

[MAR01] Marian, A., S. Abiteboul, L. Mignet. "Change-Centric Management of Versions in an XML Warehouse." Presented at *27th International Conference of VLDBs.* Rome, Italy. 11-14 September 2001. 15 May 2002 <http://www.vldb.org/conf/2001/P581.pdf>.

[MAC00] MacDonald, J. "File System Support for Delta Compression." *Master of Computer Science Thesis at University of California at Berkeley, Department of Electrical and Computer Sciences.* Berkeley California. 2000. 15 May 2002 <http://www.sourceforge.net/projects/xdelta> and <http://citeseer.nj.nec.com/rd/0%2C312806%2C1%2C0.25%2CDownload/http%253A%252F%252Fciteseer.nj.nec.com/cache/papers/cs/14402/http%253AzSzzSzwww.cs.berkeley.eduzSz%257EjmacdzSzxdfs.pdf/macdonald00file.pdf>.

[MEY86] Myers, E. "An O(ND) difference algorithm and its variations." *Algorithmica*, 1 (1986): 251-266.

[NOB97] Noble, B., M. Satyanarayanan, D. Narayanan, J. Tilton, J. Flinn, and K. Walker. "Agile application-aware adaptation for mobility". *Operating Systems Review (ACM) 51,* 5 (December 1997): 276-287.

[OPE02] OpenOffice.org. "OpenOffice.org Source Project." 15 May 2002 <http://www. openoffice.org/>.

[POO99] Poon, T., F. Curbera, D. Epstein. "Efficient encoding of XML updates." Presented at *Third Annual GCA XML Developer's Conference*, Montreal, Canada. 19-20 August 1999. 15 May 2002 <http://ibiblio.org/bosak/conf/xmldev99/curbera/curbera.pdf>.

[SCH01] Scholtz, J. "Ubiquitous Computing in the Military Environment." Presented at *SPIE: Aerosense 2001*. Orlando, Florida, 17-19 April 2001. 15 May 2002 <http://www.darpa.mil/ipto/research/uc/SPIE-f.pdf>.

[SHA89] Shasha, D. and K. Zhang. "Fast Parallel Algorithms for the Unit Cost Editing Distance Between Trees". Presented at the *ACM Symposium on Parallel Algorithms and Architectures*. Santa Fe, New Mexico. 18-21 June 1989. 15 May 2002 <http://www.cs.queensu.ca/TechReports/Reports/1995-372.ps>.

[SOU02] SourceGear Corporation. "AbiWord: Word Processing for Everyone." 15 May 2002 <http://www.abisource.com/>.

[TRI96] Tridgell, A., P. Mackerras. "The rsync algorithm." In *ANU Computer Science Technical Reports - TR-CS-96-05.* June 1996. 15 May 2002 <http://cs.anu.edu.au/techreports/1996/TR-CS-96-05.ps.gz>.

[WAL00] Walsh, N. "Making all the difference." *Sun Microsystems Technical Report.* XML Technology Center, Menlo Park, California. February 2000. 15 May 2002 <http://www.sun.com/xml/developers/diffmk>.

[WWW00] Word Wide Web Consortium. Extensible Markup Language (XML) 1.0 (Second Edition), W3C Recommendation, 6 October 2000. 15 May 2002 <http://www.w3.org/TR/2000/REC-xml-20001006>.

[WWW01] Word Wide Web Consortium. Extensible Stylesheet Language (XSL) 1.0, W3C Recommendation, 15 October 2001. 15 May 2002 <http://www.w3.org/ TR/xsl/>.

[ZHA89] Zhang, K. and D. Shasha. "Simple fast algorithms for the editing distance between trees and related problems." *SIAM Journal of Computing*, 18 (1989): 1245-1262. 15 May 2002 <http://citeseer.nj.nec.com/cache/papers/cs/16386/ http:zSzzSzwww.cs.brown.eduzSzpeoplezSzpnkzSzprojectszSzvisionzSz..zSz..zS zpaperszSzesa-treedist.pdf/klein98computing.pdf>.

[ZHA01] J. Zhang. "Mobile Data Service: Architecture, Design, and Implementation." Doctoral dissertation presented to University of Florida Department of Computer & Information Science & Engineering. Gainesville, Florida. 2002.

BIOGRAPHICAL SKETCH

Michael Lanham was born in Lewiston, New York, the United States of America. He received his Bachelor of Science (BS) in computer science from the North Carolina State University (NCSU) Department of Computer Science. He also received his BS in computer engineering from the NCSU Department of Electrical and Computer Engineering.

He received his commission as an Infantry Second Lieutenant in the United States Army through the NCSU Reserve Officer Training Corps (ROTC) department. He completed Infantry Officer's Basic Course and Ranger School and then became a platoon leader in the ”Can Do” 2nd Battalion, 15th Infantry Regiment, 3rd Infantry Division, Schweinfurt, Germany. While with 2-15 Infantry, his battalion served a six-month rotation with the United Nations Protection Force (UNPROFOR) in the Former Yugoslav Republic of Macedonia (FYROM) as Task Force Able Sentry. Michael then became the aide de camp to Commander, Special Operations Command Europe, Headquarters, United States European Command, Stuttgart, Germany. While he was at Stuttgart, his unit participated in peacekeeping missions in the Balkans as part of Operation Joint Endeavor, Embassy evacuation operations in Liberia, West Africa, and recovery operations in Croatia for Commerce Secretary Brown's airplane crash.

Michael returned to the United States to attend Infantry Officer's Advanced Course and Combined Arms Services Staff School. He joined Headquarters Company, 1st Brigade, 101st Airborne Division (Air Assault), Fort Campbell, Kentucky then took command of Delta Company, 1st Battalion, 327th Infantry Regiment. Delta Companies in the 101st Airborne Division (ABN DIV) and 82 ABN DIV are Heavy Anti-Armor Companies composed of 87 men, 28 HMMWVs, 20 TOW anti-tank missiles, 10 .50 cal machine guns, and 10 40mm automatic grenade machine guns. His battalion had multiple rotations to the National Training Center in California and the Joint Readiness Training Center in Louisiana. The Army selected him for Advanced Civil Schooling and assigned him to the University of Florida to earn a Master of Engineering in computer science. Upon graduation, he will teach computer science at the United States Military Academy.

Michael is married to a wonderful Irish-woman and has an incredibly adventurous and happy 1-year-old son.